UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea in

INGEGNERIA INFORMATICA

Tesi di Laurea

Analisi e implementazione di gerarchie di memoria a tempo d'accesso non uniforme per sistemi multicore

Candidato:

Giuseppe Serano

Relatori:

Prof. Cosimo Antonio Prete

Ing. Pierfrancesco Foglia

Ing. Marco Solinas

Anno Accademico 2010-2011

Ai miei genitori.

Indice generale

Capitolo 1

Introduzione

Con l'aumentare del livello di integrazione sui chip, i sistemi multiprocessore sono passati dalle implementazioni di sistemi multi-chip a sistemi multi-core su singolo chip (Chip Multi-Processors o CMPs [1,2]) che integrano al loro interno anche gerarchie più o meno complicate di memorie cache.

Per raggiungere prestazioni ottimali tali sistemi necessitano di una comunicazione ad elevata banda e bassa latenza sia tra processore e processore che tra gli stessi e le memorie cache. Per cui, integrare piccole cache private di alto livello vicino a ciascun core elaborativo, fornisce a ciascun processore un accesso rapido alle istruzioni ed ai dati più frequentemente utilizzati. Tali cache però, essendo di piccole dimensioni, soddisfano solo in parte la località spaziale e temporale dei programmi che devono essere eseguiti.

Quando si scatena una *miss* in una delle suddette cache, bisogna effettuare un acceso in memoria principale caratterizzato da bassa banda ed elevata latenza. Per cui occorre integrare nel chip gerarchie di memorie cache condivise di grosse dimensioni (1 Mbytes o superiori) in grado di memorizzare un grande quantità di dati e/o istruzioni, in modo da limitare, durante l'elaborazione, gli accessi in memoria principale.

Le suddette gerarchie di memoria, essendo di grandi dimensioni, saranno suddivise in banchi. Tali banchi verranno disposti in modo da occupare la minore area possibile all'interno del chip; per cui, con molta probabilità, in un determinato istante *t*, l'istruzione o il dato richiesto da un determinato processore X, si verrà a trovare in un banco N posizionato lontano dal processore richiedente. Questo provoca l'aumentare della latenza con la conseguente riduzione delle prestazioni dovuto al ritardo di propagazione dei segnali sui fili (*wire delay*, §1.1).

Nella gestione delle gerarchie di memoria cache bisogna quindi tenere conto i seguenti 2 aspetti:

- Le cache private, di piccole dimensioni e situate nei pressi del processore, riducono la latenza media di accesso, facendo migrare e replicare le copie dei blocchi di memoria vicino al processore richiedente, a discapito dell'effettiva capacità che scatena un numero elevato di *miss* che devono essere risolte Off-Chip.

- Le cache condivise, di grandi dimensioni, organizzate in sotto-banchi, minimizzano gli accessi in memoria principale, ma presentano elevate latenze dovute al ritardo di propagazione sui fili, necessario ai segnali di controllo per raggiungere la via più distante.

Con l'obiettivo di avere sistemi di elaborazione basati su gerarchie di memorie cache condivise di grande capacità che siano anche in grado di mascherare gli effetti del ritardo di propagazione dei segnali sui fili, sono state proposte architetture di cache non convenzionali, caratterizzate dall'avere un tempo medio di accesso non uniforme (Non-Uniform Cache Access, NUCA).

Una NUCA è una memoria cache, il cui spazio di *storage* è organizzato in banchi indipendenti, accessibili tramite un'infrastruttura di comunicazione scalabile. Il tempo di accesso ad una NUCA è funzione della *distanza fisica* tra il processore richiedente ed il banco che contiene il blocco di cache indirizzato. Grazie alla proprietà di non uniformità del tempo di accesso, ed adottando politiche di gestione dei dati che determinano un piazzamento ottimale dei blocchi all'interno della cache, è possibile ottenere un mascheramento degli effetti negativi che il crescente ritardo sui fili On-Chip ha sulle prestazioni totali del sistema processore-memoria (§1.2).

1.1 Wire delay

All'aumentare del livello di integrazione all'interno di un chip di silicio, si assiste alla riduzione delle dimensioni dei vari componenti (tipicamente, transistor e fili). Poiché la resistenza di una linea di trasmissione dipende in modo inversamente proporzionale alla sezione della linea stessa, la riduzione delle dimensioni dei fili (in particolare, della loro sezione) comporta un aumento della loro resistenza. Di conseguenza si ha un aumento del ritardo di propagazione dei segnali elettrici che è direttamente proporzionale al prodotto RC, poichè la capacità equivalente del filo non diminuisce alla stessa velocità con cui la sua resistenza aumenta. Questo fenomeno è

particolarmente significativo all'aumentare della lunghezza della linea di trasmissione, mentre ha un impatto trascurabile in caso di linee corte. Dal momento che le frequenze di lavoro dei moderni sistemi di elaborazione sono in continua crescita, il numero di cicli di clock che vengono spesi affichè un segnale si propaghi aumenta; tale aumento, unito a quello del ritardo sui fili, affligge significativamente le prestazioni dei sistemi. Se si considera ad esempio una cache da 16 Mbyte per una tecnologia a 50 nm, il banco più vicino può essere acceduto in 4 cicli di clock, mentre per il più lontano si può arrivare anche a 50 cicli di attesa. Questo fenomeno è noto in letteratura come wire delay [12].

1.2 NUCA

Per ridurre il numero di *miss* che devono essere risolte Off-Chip bisogna integrare nel sistema CMP una cache condivisa avente una grande capacità di *storage*. Una cache di grosse dimensioni è caratterizzata da un tempo di accesso estremamente sensibile al *wire delay*.

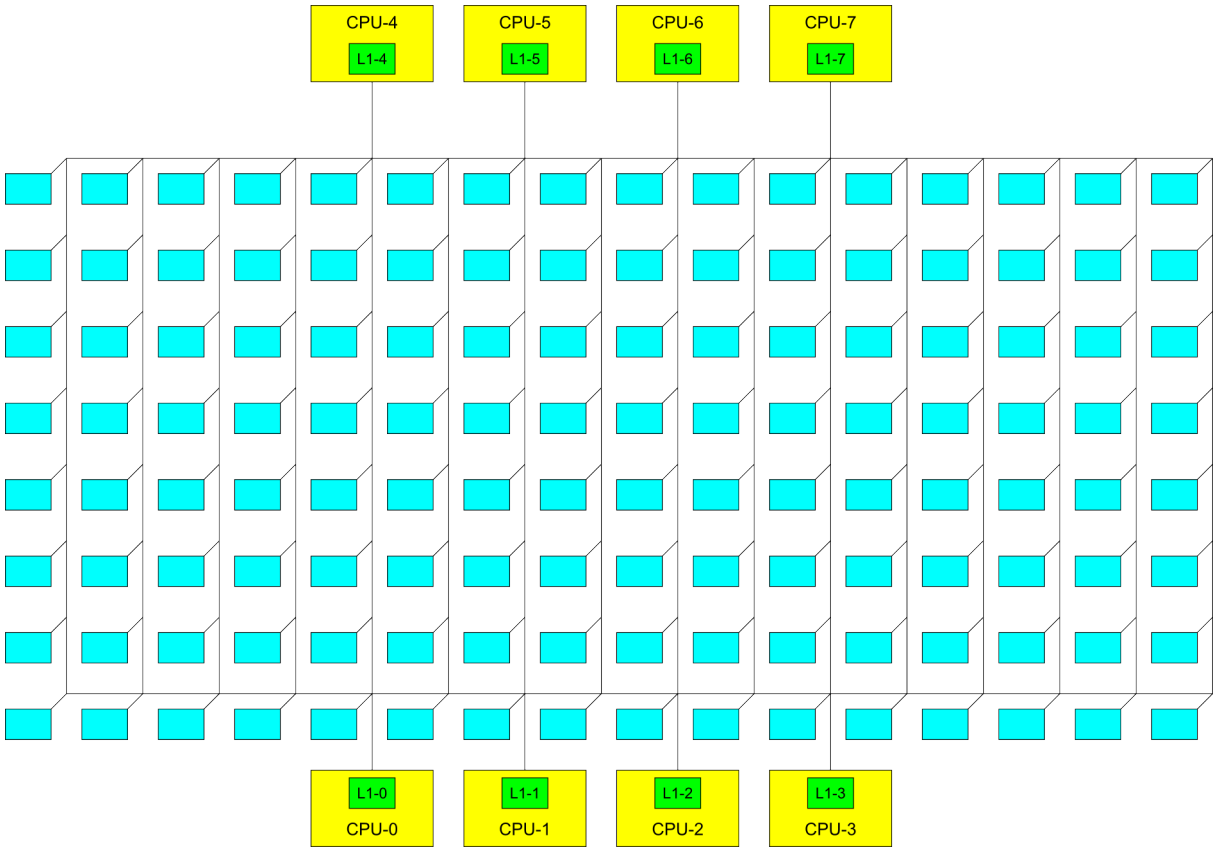

Figura 1 Architettura CMP basata su cache NUCA

Le cache NUCA [13, 14] sono state introdotte allo scopo di poter combinare i benefici in termini di *miss rate*, dovuti all'elevata capacità della cache, con un tempo medio di accesso ridotto rispetto al caso di una cache tradizionale di pari dimensioni. Questo obiettivo è raggiunto grazie ad una organizzazione in cui i banchi sono connessi fra di loro mediante un'infrastruttura di comunicazione scalabile, tipicamente una *Network-on-Chip* (NoC) [15, 16]. Una NoC è composta da link e switch, e prevede un paradigma di comunicazione a scambio di messaggi. In Figura 1 è illustrato un sistema CMP ad 8 processori, in cui l'ultimo livello di cache è una NUCA, e l'infrastruttura di comunicazione è una NoC.

Quando si deve accedere ad una NUCA per poter riferire un blocco, è necessario sapere quanti e quali banchi bisogna "interrogare" per poter soddisfare la richiesta. A tale scopo, sono state definite due differenti *politiche di mapping*:

- **Statico**: dato l'indirizzo fisico di un blocco, esiste un unico banco della NUCA che può contenere una copia del blocco. Una NUCA che adotta questo tipo di politica di mapping è detta *Static-NUCA* (**S-NUCA**);

- **Dinamico**: dato l'indirizzo fisico di un blocco, esiste un insieme definito di banchi della NUCA che può contenere una copia del blocco; tale insieme di banchi è chiamato *bankset*. Una NUCA che adotta questo tipo di politica di mapping è detta *Dynamic-NUCA* (**D-NUCA**).

In una S-NUCA, una richiesta per un blocco è ricevuta solo dal banco sul quale è mappato l'indirizzo fisico. Se il blocco è presente nel banco, si avrà una *hit*, altrimenti si avrà una *miss*.

Dal momento che i banchi più vicini al processore richiedente sono anche quelli verso cui esso sperimenta un tempo di accesso più basso, è ragionevole pensare di *spostare* i blocchi più frequentemente acceduti nei banchi a minor latenza. In una D-NUCA, un blocco può *migrare* da un banco all'altro in funzione della *frequenza di accesso* da parte dei processori, con l'intento di spostarsi verso i banchi più vicini al processore che lo riferisce più spesso; la migrazione di un blocco può avvenire solo tra banchi che appartengono allo stesso *bankset*.

Sia la S-NUCA che la D-NUCA mascherano gli effetti del *wire delay*. Infatti, una cache tradizionale deve sempre aspettare il tempo di risposta della via più lontana, mentre nelle NUCA il

tempo di accesso dipende dalla distanza fisica del banco rispetto al processore richiedente. In particolare, nella D-NUCA i blocchi più frequentemente acceduti tenderanno ad essere memorizzati nei banchi vicini, mentre quelli acceduti più raramente saranno contenuti nei banchi più lontani.

Schemi più avanzati prevedono la possibilità di introdurre un sistema di replicazione (si parla di Re-NUCA) di determinate categorie di blocchi, in modo da avere più copie indipendenti che si spostano dinamicamente con l'obiettivo di piazzarsi nella posizione più vicina ai richiedenti.

1.3 Lavoro di tesi

Il presente lavoro di tesi si propone di studiare il trade-off tra una Re-NUCA, che adotta un sistema ottimizzato di replicazione in grado di tenere sottocontrollo il *miss rate*, ed una configurazione S-NUCA a cache private, che qui chiameremo Private S-NUCA (PS-NUCA), in cui ogni core mantiene una propria copia locale nella cache di ultimo livello. A tale scopo, nel presente lavoro è stata progettata, realizzata e testata l'architettura di gerarchie di memoria PS-NUCA, in cui i banchi di una S-NUCA vengono esplicitamente suddivisi e mappati come privati di ciascun processore. Inoltre, per gestire la coerenza, è stato progettato un protocollo basato sul MESI per sistemi a Directory in cui la Directory stessa risiede in memoria principale. Per limitare gli accessi Off-Chip alla Directory, è stata inserita nell'architettura un'ulteriore livello di cache (Directory Cache) che memorizza solo i blocchi di directory associati ai blocchi di memoria.

I risultati sperimentali, ottenuti per via simulativa, vengono analizzati e confrontati con architetture ti tipo S-NUCA, D-NUCA e Re-NUCA, al fine di valutare i comportamenti delle varie configurazioni dal punti di vista delle prestazioni e della scalabilità.

Il presente lavoro è organizzato come segue: il capitolo 2 descrive gli strumenti di sviluppo e testing che sono stati utilizzati. Il capitolo 3 introduce il sistema CMP PS-NUCA considerato. Nel capitolo 4 è illustrato il protocollo di coerenza basati sul MESI per sistemi a Directory. Nel capitolo 5 sono presentati e discussi i risultati ottenuti nelle simulazioni. Il capitolo 6 conclude il lavoro ed introduce i possibili sviluppi futuri. In appendice sono infine riportati l’elenco degli eventi e delle transizioni che regolano il comportamento della PS-NUCA realizzata.

Capitolo 2

Ambiente di sviluppo e test

In questo capitolo viene presentato Ruby, un simulatore di gerarchie di memorie che consente di simulare dettagliatamente le architetture di cache di qualunque sistema di elaborazione, sia esso mono-processore o multi-processore, SMP o CMP, a cache privata o cache condivisa. In particolare, si vedrà come Ruby gestisce gli eventi che possono essere scatenati nel sistema, e quali strumenti mette a disposizione dello sviluppatore per descrivere in maniera semplificata la macchina a stati dei controllori delle cache e delle memorie principali. Infine, verrà illustrato anche il funzionamento del tester, un utile strumento che fa parte della suite GEMS (di cui Ruby è parte fondamentale) e che consente di eseguire il debugging dei protocolli implementati.

2.1 Ruby

Ruby è un simulatore di gerarchie di memorie per sistemi di elaborazione. In particolare, Ruby modella memorie cache, controllori di memorie cache e reti di interconnessione tra i componenti della gerarchia di memoria. Ruby combina la simulazione temporale di componenti che risultano in gran parte indipendenti dal protocollo di coerenza con la possibilità di specificare i componenti che invece dipendono dal protocollo di coerenza stesso (ad esempio, controllori della memoria cache). I componenti di Ruby indipendenti dal protocollo di coerenza includono la rete di interconnessione, le cache, la memoria, i buffer di messaggi ed un insieme di elementi che fungono da “collante” per essi. Ruby modella sia la gerarchia di cache private per ciascun processore, sia la gerarchia di cache condivisa tra tutti i processori di un sistema. Molte caratteristiche della cache, come ad esempio la dimensione e l’associatività, sono parametri configurabili. Per quanto riguarda i componenti dipendenti dal protocollo di coerenza (ad esempio i controllori della cache), essi vengono descritti in **SLICC** (*Specification Language for Implementing Cache Coherence*), un linguaggio pensato per specificare la macchina a stati dei protocolli di coerenza in maniera semplificata.

2.2 Architettura software di Ruby

Ruby è implementato nel linguaggio C++ ed utilizza il modello di code ad eventi per simulare il timing. I componenti comunicano tra di loro utilizzando buffer di messaggi con banda e latenza assegnata; il componente che legge da un determinato buffer viene “svegliato” nel momento in cui il prossimo messaggio è disponibile ad essere prelevato dal buffer. Molti buffer vengono gestiti utilizzando la politica *first-in-first-out* (FIFO), ma è possibile specificare anche altri tipi di politica. La simulazione procede invocando il metodo *wakeup*() per il prossimo evento schedulato dalla coda degli eventi.

Come è stato asserito in precedenza, Ruby è un simulatore ad eventi discreti. Le classi principali per la realizzazione di tale simulazione sono `Consumer`, `MessageBuffer`, `EventQueue` ed `EventQueueNode`. Nel file *Consumer.c* è contenuta l’implementazione della classe `Consumer`. Tale classe è di tipo virtuale pura e costituisce la classe base per tutte le classi mediante le quali vengono implementati gli oggetti che nel corso della simulazione sono dei consumatori di eventi. In particolare, tra tali oggetti si trovano i controllori della memoria cache e gli elementi che realizzano la rete di comunicazione. Per comodità, designeremo col generico termine “consumer” qualsiasi oggetto la cui classe è derivata dalla classe base `Consumer`.

La classe `MessageBuffer` realizza le code di messaggi che ciascun consumer deve gestire; ogni consumer ha una coda per i messaggi in ingresso ed una o più code per i messaggi in uscita (Figura 2).

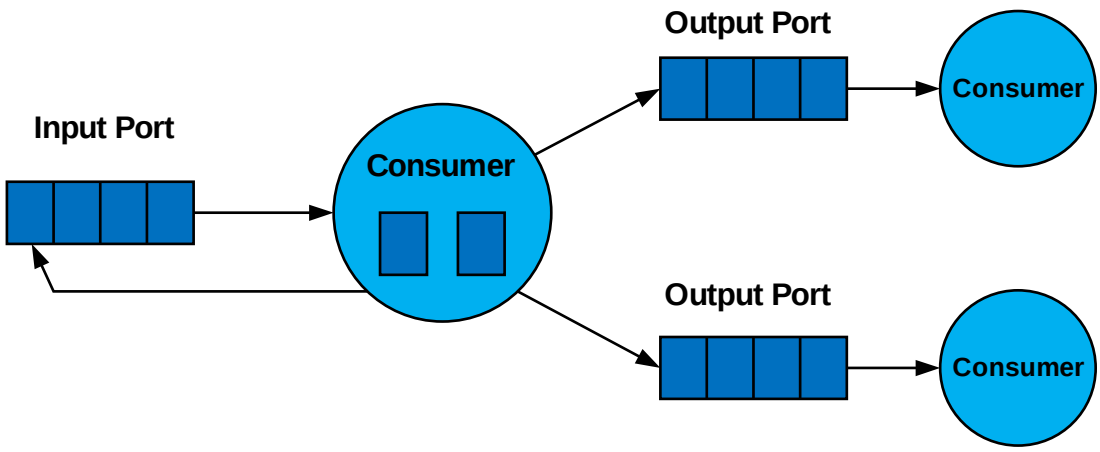

Figura 2 Consumer e code di messaggi

Nei file *EventQueue.c*, *EventQueue.h*, *EventQueueNode.c* ed *EventQueueNode.h* è contenuto il codice che realizza gli eventi e che implementa l'engine per la gestione della coda globale degli eventi e lo scheduling degli eventi stessi.

La coda globale degli eventi viene implementata mediante la classe `EventQueue`; i dati membro principali di tale classe sono il puntatore *m_prio_heap_ptr*, puntatore alla coda degli eventi, e *m_global_time*, che ad un certo istante della simulazione rappresenta il tempo attuale raggiunto.

L'evento viene invece implementato mediante la classe `EventQueueNode`; i dati membro principali di tale classe sono *m_consumer_ptr*, puntatore all'oggetto di tipo Consumer che sarà "consumatore" dell'evento, e *m_time*, tempo assoluto a cui l'evento stesso dovrà essere gestito. Un evento viene creato mediante la funzione:

```
void EventQueue::scheduleEventAbsolute(Consumer* c, Time t)
```

Ad ogni evento viene quindi assegnato sia un tempo t a cui schedulare l'evento stesso, sia un consumatore: l'engine provvede quindi a far avanzare il tempo t e a schedulare gli eventi in base al tempo assegnato; ogni evento può in generale generare altri eventi che vengono inseriti nella coda globale.

Il trigger degli eventi viene implementato mediante la funzione:

```
void EventQueue::triggerEvents(Time t)
```

la quale scorre la coda degli eventi selezionando tutti quelli aventi tempo di scheduling inferiore o uguale a t ed invocando il metodo `wakeup()` sull'oggetto di tipo "consumer" relativo a tale evento. Complessivamente, quindi, la simulazione avanza ciclicamente nel modo seguente:

1) Un consumatore viene svegliato ed effettua la scansione della porta di messaggi in ingresso;
2) Un consumatore compie delle azioni e cambia il proprio stato sulla base del tipo e del contenuto dei messaggi in ingresso;
3) Un consumatore accoda nuovi messaggi nelle sue porte di uscita; tali messaggi verranno prelevati successivamente dalle porte in ingresso da parte di altri consumatori.

2.3 La rete di Interconnessione

La rete di interconnessone è un substrato unificato utilizzato per modellare la comunicazione all'interno della gerarchia di cache e tra la cache ed i controllori della memoria. Tale modello monolitico viene utilizzato per simulare qualsiasi tipo di comunicazione, anche tra controllori che in un sistema CMP simulato sono situati sul medesimo chip. Questo tipo di scelta offre sufficiente flessibilità per simulare il timing di qualsiasi tipo di sistema di interconnessione.

Nel modello in questione, i controllori comunicano tra di loro inviandosi dei messaggi, ed il sistema di interconnessione tiene traccia della temporizzazione che caratterizza il flusso dei messaggi attraverso la rete. I messaggi aventi più di un destinatario (ad esempio nella comunicazione di tipo broadcast) si avvalgono di efficienti algoritmi di routing multicast.

La rete di interconnessione modellata da Ruby è di tipo punto-punto in cui i punti di interconnessione fra nodi distinti sono degli switch, e può essere configurata in modo analogo a quelle incluse nei sistemi multiprocessore più diffusi, ad esempio i sistemi directory-based e snooping-based. Per simulare i sistemi con protocollo basato su directory, Ruby supporta tre diversi tipi di reti non ordinate: una rete semplificata punto-punto completamente connessa, una rete a routing dinamico con architettura a toro bidimensionale (ispirata a quella dell'Alpha 21364) e una rete definibile a completa discrezione dell'utente.

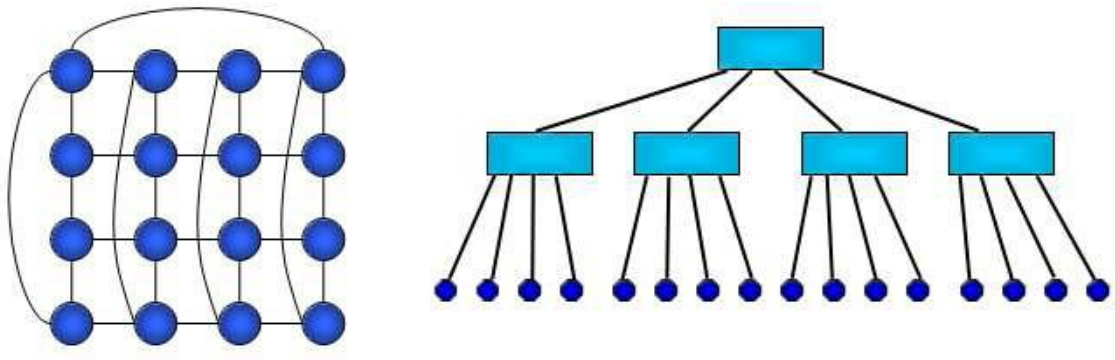

Figura 3 Topologia a toro bidimensionale e a gerarchia di switch

Le prime due reti sono generate automaticamente utilizzando determinati parametri di configurazione, mentre la terza è una rete arbitraria che l'utente può realizzare modificanto un apposito file di configurazione. Quest'ultima caratteristica consente all'utente di descrivere reti anche piuttosto complicate, come ad esempio una rete per le PS-NUCA. Per i sistemi basati sullo snooping, Ruby include due tipi di reti totalmente ordinate: una rete a crossbar ed una rete a gerarchia di switch. Entrambe le reti utilizzano una gerarchia di uno o più switch per creare un ordine totale di richieste di coerenza alla radice della rete.

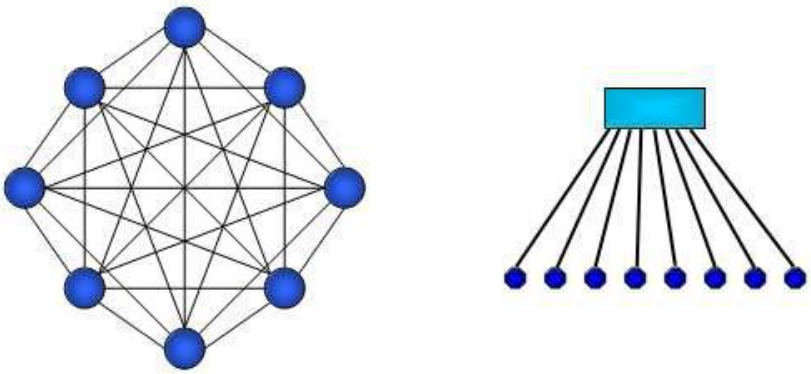

Figura 4 Topologia punto-punto completamente connessa e a crossbar

La topologia delle interconnessioni è specificata attraverso un insieme di collegamenti tra gli switch, e le tabelle di routing vengono calcolate ad ogni esecuzione, permettendo di aggiungere di volta in volta al sistema nuove topologie. Ogni link della rete ha una banda limitata, ma la rete di interconnessione non modella ulteriori dettagli del livello fisico. Di default gli switch hanno buffer di dimensione illimitata, ma Ruby supporta per certe reti anche buffer finiti.

2.3.1 Rete definita dall'utente

Esiste per l'utente la possibilità di specificare completamente l'architettura di una rete di interconnessione. Ciò si ottiene impostando come segue uno dei parametri di configurazione di Ruby:

```
ruby0.setparam_str_g_NETWORK_TOPOLOGY FILE_SPECIFIED
```

La rete viene costruita dal simulatore a partire da uno dei file con estensione *.txt* contenuti nella directory *[GEMS]/ruby/network/simple/Network_Files*. Più in dettaglio, il simulatore sceglie automaticamente uno dei file presenti in tale directory in base ad altri parametri di configurazione. Ad esempio il file:

```
NUCA_Procs-8_ProcsPerChip-8_L2Banks-256_Memories-4.txt
```

Viene scelto dal simulatore nel caso in cui la cache sia di tipo NUCA, il numero totale dei processori sia 8, il numero di processori per chip sia 8, il numero totale di banchi della L2 sia 256 ed il numero totale di memorie principali sia 4.

Un file di descrizione della rete di interconnessione inizia sempre con quattro righe analoghe alle seguenti:

```
processors:8
procs_per_chip:8
L2banks:256
memories:8
```

Tali righe specificano i valori dei quattro parametri analogamente a quanto avviene per il nome del file stesso. Nelle righe successive del file sono elencati tutti i link tra i nodi che costituiscono la rete, uno per riga. I nodi possono essere di tipo interno, ovvero gli switch della rete, o di tipo esterno. Tra quelli di tipo esterno si trovano le cache L1 ed i banchi della cache L2.

Ad esempio la seguente riga:

```
ext_node:L1Cache:0 int_node:1 link_weight:1 link_latency:1 bw_multiplier:72
```

realizza un collegamento tra un nodo esterno, costituito dal controller di una cache L1 avente identificatore 0, ed un nodo interno, ovvero uno switch della rete avente identificatore 1. A tale link vengono associati un peso, una latenza (link_latency) ed una banda (bw_multiplier).

Come ulteriore esempio la seguente riga:

```
int_node:0 int_nod:8 link_weight:57 link_latency:1 bw_multiplier:32
```

realizza invece un collegamento tra due nodi interni, ovvero tra due switch della rete di interconnessione.

2.3.2 Implementazione della rete di interconnessione

Gli elementi che compongono il sistema vengono allocati all'interno del costruttore della classe `Chip`, implementato nel file *Chip.c*. La rete di interconnessione viene costruita dal costruttore della classe `Topology`; nell'ipotesi di architettura di rete definita dall'utente, i link vengono allocati scandendo il file in cui è descritta la rete e creando per ciascuno di essi, mediante la funzione `addLink()`, un elemento dei vettori di interi *m_links_src_vector* e *m_links_dest_vector*; tali vettori sono tali per cui gli interi *m_links_src_vector[i]* e *m_links_dest_vector[i]* rappresentano due nodi della rete collegati da un link; è da sottolineare il fatto che all'interno di tali vettori i nodi non sono rappresentati con lo stesso identificatore con cui si trovano nel file, in quanto se così fosse alcuni nodi distinti avrebbero il medesimo identificatore (ad esempio, il banco di cache L2 con identificatore 2 sarebbe indistinguibile dallo switch avente lo stesso identificatore). Più in dettaglio, l'identificatore con cui un elemento della rete entra a far parte di questi due vettori viene costruito sommando all'identificatore dell'elemento un offset che dipende dal tipo di elemento; in particolare, alcuni offset importanti sono:

```
L1_Cache:  0
L2_Cache:  Numero di processori
  Switch:  Numero di processori +
                Numero di banchi di cache L2 +
                Numero di banchi di RAM
```

Successivamente vengono creati i vettori `m_links_latency_vector`, `m_links_weight_vector` e `m_bw_multiplier_vector`, in base rispettivamente alla latenza, al peso e alla banda di ciascun link; infine viene chiamata la funzione `makeLink()` per ogni coppia di switch: tale funzione esplora l'intera topologia della rete e costruisce un oggetto di

tipo `NetDest` che rappresenta il path più vantaggioso per inviare un pacchetto tra i due switch. Al termine di questa procedura, gli oggetti rilevanti che restano allocati sono i dati membro della classe `SimpleNetwork` `m_toNetQueues` e `m_fromNetQueues`; tali oggetti, che sono matrici di puntatori a `MessageBuffer`, non solo rappresentano i buffer di messaggi, ma incapsulano anche le informazioni per il routing dei pacchetti a seconda di quale sia lo switch destinatario sulla base dei risultati delle chiamate alla funzione `makeLink()`.

2.4 L'ambiente di testing di Ruby

Ruby offre anche la possibilità di effettuare delle *tracedriven simulation*. Visto che lo scopo principale del simulatore è quello di costruire un modulo per una *full-system simulation* (e quindi *execution driven*), questa possibilità può essere utile tipicamente per testare la correttezza di un protocollo di coerenza; inoltre, effettuando un debugging del tester (utilizzando ad esempio ddd, un'interfaccia grafica per gdb) dopo aver compilato Ruby con le opzioni di debugging, è possibile seguire più da vicino il flusso di esecuzione della simulazione prescindendo dalla fullsystem simulation in modo tale da soffermarsi sul codice di Ruby stesso, per comprenderlo meglio ed eventualmente modificarlo.

Il codice relativo alla trace-driven simulation è contenuto nei file *Tracer.c* e *TraceRecord.c*; al termine della compilazione, oltre al modulo Ruby-v9.so linkabile a Simics, viene creato anche l'eseguibile *tester.exec* sulla base del codice di tali file. Tale eseguibile può essere lanciato da shell con il comando:

```
tester.exec –p 8 –l length –s num –z tracefile_name
```

Dove l'opzione `–p` (necessaria per avviare correttamente il tester) è seguita dal numero di processori che compongono il sistema, l'opzione `–l` specifica la lunghezza del test in termini di numero di istruzioni eseguite complessivamente dai `p` processori passati all'ambiente con il primo parametro; l'opzione `–s` specifica il punto in cui iniziare a scrivere sullo standard output tutta l'evoluzione del sistema in termini di tuple del tipo *{indirizzo, nodo, evento, stato_iniziale > stato_successivo}* che si verificano in ogni nodo del sistema, mentre la stringa che segue l'opzione `–z` è il nome del file che contiene la traccia. Il file della traccia è dunque una sequenza di righe ciascuna delle quali rappresenta un riferimento in memoria. Ciascuna riga è del tipo:

```
ProcID     HexAddress      0     ReferenceType
```

Il primo elemento è un intero che coincide con l'identificatore del processore che effettua il riferimento, il secondo elemento è un valore esadecimale che coincide con l'indirizzo di memoria riferito, il terzo elemento deve essere posto a zero ed il quarto elemento è una stringa che rappresenta il tipo di operazione che si attua con il riferimento. Ad esempio la seguente riga:

```
3   0x00900001      0          LD
```

rappresenta un riferimento effettuato dal processore di identificatore 3 all'indirizzo di memoria 0x00900001; il riferimento è una LOAD. Il tester legge sequenzialmente la traccia e dopo la lettura di ciascun riferimento genera gli eventi che verranno gestiti e simulati. L'opzione `-z` può anche essere omessa: in questo caso, il tester genera automaticamente un file di traccia di riferimenti in memoria, e questa traccia è costruita in modo che l'esecuzione corrispondente abbia un elevato grado di *sharing*, con molte scritture sui blocchi intervallate da delle letture. In questo modo, il tester è in grado di sollecitare tutte le corse critiche del protocollo che si intende testare, così da individuare possibili errori sia sintattici, come ad esempio l'utilizzo di oggetti o campi non allocati, sia semantici, quali ad esempio situazioni di *deadlock* e *livelock*. Come avremo modo di spiegare più avanti, durante la fase di test è opportuno lasciare che sia il tester a generare la traccia dell'esecuzione da simulare: infatti, per quanto "patologiche" siano le esecuzioni da esso generate, rappresentano comunque delle situazioni possibili, per quanto altamente improbabili, che si possono sempre verificare.

I parametri con cui viene avviato il tester possono essere modificati. Infatti, oltre alle opzioni che vengono passate quando viene lanciato l'eseguibile, è possibile configurare il tester in modo da specificare quali siano le dimensioni delle cache, del blocco, e se la rete di interconnessione deve essere generata automaticamente o passata da file. Per configurare questi aspetti, è sufficiente modificare il file *tester.default*, contenuto della cartella *[GEMS]/ruby/config*. Notare che in quella stessa cartella esiste il file *rubyconfig.default*, che serve in generale per configurare Ruby: anche il tester legge i valori contenuti in quest'ultimo file, ma successivamente legge i valori dei parametri dal file *tester.default*, sovrascrivendo di fatto i valori contenuti in *rubyconfig.default*. Da segnalare sono due parametri, che in *rubyconfig.default* sono impostati entrambi a *false*:

`g_DETERMINISTIC_DRIVER`
`RANDOMIZATION`

Nel file *tester.default*, il primo parametro è impostato di default al valore *false*, mentre il secondo al valore *true*. Se `g_DETERMINISTIC_DRIVER` viene settato, il tester genererà una traccia di esecuzione semplificata, utile per semplici verifiche di "sanità" ma anche per effettuare tuning dei parametri. Più critico è invece il comportamento quando il parametro `RANDOMIZATION` vale *true*: infatti, questo comporta che Ruby (attenzione, non solo il tester) inserisca dei ritardi random sull'inoltro dei messaggi (ovvero, sull'istante in cui i messaggi lasciano le code di uscita), allo scopo di forzare un numero maggiore di corse critiche. Se di per sé questo potrebbe essere utile, ci possono essere casi in cui è importante che anche in fase di test una tale condizione non si verifichi (ad esempio, se è importante garantire l'ordinamento degli istanti di servizio dei messaggi anche per *virtual network* che non prevedono ordinamento punto-punto delle richieste - § 2.5): per questo, è possibile mettere a *false* il parametro, agendo sul file *tester.default*.

2.5 Specifica di un protocollo

La specifica di un protocollo (sia esso di coerenza, oppure più complesso) viene fatta utilizzando il linguaggio SLICC. Questo linguaggio consente di descrivere sia la struttura che il comportamento della macchina a stati finiti di un controllore di memoria, sia essa una cache (di qualunque livello della gerarchia) che la memoria principale stessa. In realtà, è possibile descrivere anche delle macchine a stati che non sono controllori di memoria, purché rappresentino dei nodi del sistema che Ruby gestisce (ovviamente, questi nodi devono anch'essi essere attaccati alla rete di interconnessione): questo aspetto, però, prescinde dagli scopi di questa tesi.

La struttura generale di un file contenente codice SLICC (estensione *.sm*) è la seguente:

```
machine (nomeMachine, desc = "Un commento") {

  MessageBuffer di input;  // uno per ciascuna virtual network
  MessageBuffer di output; // uno per ciascuna virtual network
```

```
  enumeration (State, desc = "Un commento") {
    // elenco completo degli stati della macchina
    STATO,      desc = "Un commento";
  }

  enumeration (Event, desc = "Un commento") {
    // elenco completo degli eventi della macchina
    EVENTO,     desc = "Un commento";
  }

  // Strutture dati e funzioni di utilità

  // Dichiarazioni delle porte di uscita
  outport (nomeOutport, TipoMsg, bufferAssociato);

  // Dichiarazione e definizione delle porte di ingresso
  inport (nomeInport, TipoMsg, bufferAssociato) {
    // codice di gestione dei messaggi in ingresso
    // da qui vengono triggerati gli eventi
    if (in_msg.Type == TIPO) {
      // eventuali controlli
      trigger (Event:EVENTO, in_msg.Address);
    }
  }

  // Dichiarazione e definizione delle action
  action (a_azione, desc = "Un commento") {
    // codice SLICC
  }

  // Dichiarazione e definizione delle transition
  transition (STATO_PARTENZA, EVENTO, STATO_FINALE) {
    // sequenza di action
  }

// fine machine
}
```

Verrà discusso di seguito il significato di ciascun campo, facendo ricorso ad un esempio. Supponendo di voler descrivere il comportamento di una memoria cache quando il protocollo di coerenza è di tipo MI, ed avendo un solo livello di cache:

```
machine (L1Cache, desc = "Simple MI Protocol") {
…
}
```

2.5.1 Stati ed Eventi

Gli stati e gli eventi vengono elencati utilizzando il costrutto `enumeration`; esso ha come argomenti l'identificatore dell'insieme che si intende enumerare e una stringa (che in pratica serve solo come commento) che descrive cosa rappresenta l'insieme stesso. Gli stati sono suddivisi in *stati base* e *stati transienti*. Gli stati base sono gli stati veri e propri della macchina; gli stati transienti sono quelli in cui può trovarsi la macchina a stati nel corso di una transizione, mai a transizione completata:

```
enumeration(State, desc = "Cache states") {
  // Stable States
  I,       desc = "Not Present/Invalid";
  M,       desc = "Modified";
  // Transient States
  MI,      desc = "Modified, issued PUT";
  II,      desc = "Not Present/Invalid, issued PUT";
  IS,      desc = "Issued request for IFETCH/GETX";
  IM,      desc = "Issued request for STORE/ATOMIC";
}
```

Gli eventi sono quelli che la macchina a stati può lanciare, quando riceve un nuovo messaggio o al seguito del verificarsi di opportune condizioni. Gli eventi possono essere relativi a messaggi che arrivano dal processore (se la macchina a stati è quella della cache di primo livello) oppure dalla rete di interconnessione:

```
enumeration(Event, desc = "Cache events") {
  // From processor
  Load,              desc = "Load request from processor";
  Ifetch,            desc = "Ifetch request from processor";
  Store,             desc = "Store request from processor";
  // From the network
  Data,              desc = "Data from network";
  Fwd_GETX,          desc = "Forward from network";
  Replacement        desc = "Replace a block";
  Writeback_Ack,     desc = "Ack from directory for a writeback";
  Writeback_Nack,    desc = "Nack from directory for a writeback";
}
```

Una *transizione di stato*, quindi, è un cambiamento che parte da uno stato base a seguito dello scatenarsi di un particolare evento, e si completa quando si ritorna in un altro stato base, passando per zero o più stati transienti (§ 2.5.6).

2.5.2 Buffer di messaggi

I buffer dei messaggi sono dichiarati in SLICC come degli oggetti di tipo `MessageBuffer`. Il tipo `MessageBuffer` è una classe C++ di Ruby, definita in *[GEMS]/ruby/buffers/MessageBuffer.h*. Il costrutto utilizzato prevede la dichiarazione del tipo seguita dall'identificatore del buffer, dopo il quale, separati da virgole, ci sono una serie di parametri:

1) La parola chiave `network` seguita dalla stringa “To” o “From” a seconda che la porta relativa al buffer sia rispettivamente in uscita o in ingresso alla macchina a stati;

2) La parola chiave `virtual_network`, seguita da una stringa che contiene l’intero che rappresenta l’identificatore della *virtual network* a cui quel buffer è associato;

3) La parola chiave `ordered`, seguita dalla stringa “true” o dalla stringa “false”, a seconda che per quella *virtual network* si debba o meno forzare l’ordinamento punto-punto.

L’ordinamento per le *virtual network* viene forzato da Ruby in due modi:

i) disabilitando l’eventuale *routing adattivo* per tutti i messaggi che viaggiano sulla *virtual network* ordinata (in modo da evitare che l’ordinamento venga violato perché i messaggi seguono percorsi differenti)

ii) forzando il programmatore ad accodare sui buffer di uscita di una *virtual network* ordinata messaggi che abbiano tutti la stessa latenza di uscita (se la latenza è N, il messaggio che viene accodato all’istante t, uscirà all’istante t+N, il messaggio successivo che sarà accodato all’istante t+T uscirà all’istante t+T+N).

```
MessageBuffer   requestFromCache,
                network = "To",
                virtual_network = "0",
                ordered = "true";

MessageBuffer   responseFromCache,
                network = "To",
                virtual_network = "1",
                ordered = "true";

MessageBuffer   forwardToCache,
                network = "From",
                virtual_network = "2",
                ordered = "true";

MessageBuffer   responseToCache,
                network = "From",
                virtual_network = "1",
                ordered = "true";
```

2.5.3 Strutture dati della macchina a stati

Nella macchina a stati è possibile dichiarare delle strutture dati che saranno usate durante il funzionamento del sistema. Ad esempio, dovendo scrivere il controllore di una memoria cache, occorre dichiarare al suo interno un oggetto di tipo `CacheMemory` (si tratta di una classe C++ definita in *[GEMS]/ruby/system/CacheMemory.h*), ed eventualmente un oggetto di tipo `TBETable`, che rappresenta gli *MSHR* (anche la `TBETable` è una classe C++ definita in *[GEMS]/ruby/system/TBETable.h*).

La classe `CacheMemory` è un template. È possibile dichiarare il tipo da passare al costruttore mediante il costrutto `structure`:

```
// CacheEntry
structure(Entry, desc = "...", interface = "AbstractCacheEntry") {
  State         CacheState,     desc = "cache state";
  bool          Dirty,          desc = "data dirty";
  DataBlock     DataBlk,        desc = "data for the block";
}
```

Col costrutto `external_type` si può dare visibilità ai metodi di una classe C++ che è stata istanziata in SLICC. Con riferimento alla classe `CacheMemory`:

```
external_type(CacheMemory) {
  bool     cacheAvail(Address);
  Address  cacheProbe(Address);
  void     allocate(Address);
  void     deallocate(Address);
  Entry    lookup(Address);
  void     changePermission(Address, AccessPermission);
  bool     isTagPresent(Address);
}
```

Per allocare l'oggetto, infine, esiste il costrutto `template_hack` che specifica il tipo del template, ed il costrutto `constructor_hack` che serve per passare i parametri al costruttore. Ad esempio:

```
CacheMemory     cacheMemory,
                template_hack = "<L1Cache_Entry>",
                constructor_hack = '  L1_CACHE_NUM_SETS_BITS,
                                      L1_CACHE_ASSOC,
                                      MachineType_L1Cache,
                                      int_to_string(i)+"_L1"',
                abstract_chip_ptr = "true";
```

Notare che il `template_hack` prende come valore `L1Cache_Entry`, tra parentesi angolate e come stringa. Questo non è arbitrario: infatti è necessario che il `template_hack` sia costruito come nome della macchina (quello che viene dato all'inizio del file, nel costrutto `machine`, in questo caso `L1Cache`) seguito da un carattere di underscore, ed infine il nome che è stato dato alla entry della cache nel costrutto `structure` (in questo caso, `Entry`). Per il significato dei vari parametri del `constructor_hack`, è sufficiente analizzare il prototipo del costruttore dell'oggetto che si vuole allocare.

Un'ultima osservazione da fare riguarda le cache di primo livello: in questo caso, è necessario allocare anche un oggetto di tipo `Sequencer` (una classe C++ definita in *[GEMS]/ruby/system/Sequencer.h*), che rappresenta il componente che dialoga con il processore:

```
Sequencer           sequencer,
                    constructor_hack = "i"
                    abstract_chip_ptr = "true";
```

2.5.4 Porte di ingresso e porte di uscita

Le porte di ingresso e le porte di uscita vengono dichiarate in SLICC mediante i costrutti *inport* ed *outport*. A tutte le porte sono associati dei buffer, che devono essere preventivamente dichiarati come oggetti di tipo `MessageBuffer` (§ 2.5.2). I parametri di una *inport* e di una *outport* sono tre: il primo è l'identificatore della porta, il secondo è il tipo di messaggio che essa gestisce, il terzo è l'identificatore del buffer ad essa associata.

Le *outport* non hanno del codice ad esse associate; pertanto, saranno solamente dichiarate:

```
out_port(requestNetwork_out, RequestMsg, requestFromCache);
out_port(responseNetwork_out, ResponseMsg, responseFromCache);
```

Le *inport*, invece, hanno del codice associato, in quanto all'atto della ricezione di un messaggio bisogna decidere quale evento è necessario scatenare (sulla base sia del tipo di messaggio ricevuto che su eventuali condizioni aggiuntive, per esempio sull'eventuale azzeramento di un contatore, o sul valore di un determinato flag contenuto nelle strutture dati ad esempio, nel TBE- della macchina a stati):

```
in_port(forwardRequestNetwork_in, RequestMsg, forwardToCache) {
  if (forwardRequestNetwork_in.isReady()) {
    peek(forwardRequestNetwork_in, RequestMsg) {
      if (in_msg.Type == CoherenceRequestType:GETX) {
        trigger(Event:Fwd_GETX, in_msg.Address);
      }
      else if (in_msg.Type == CoherenceRequestType:WB_ACK) {
        trigger(Event:Writeback_Ack, in_msg.Address);
      }
      else {
        error("Unexpected message");
      }
    }
  }
}
```

Una porta di ingresso "speciale" è la `MandatoryQueue`. La cache di primo livello è connessa direttamente al processore. Le richieste che il processore fa verso il sottosistema di memoria vengono messe da Ruby proprio nella `MandatoryQueue`. Per poter allocare una `MandatoryQueue`, è sufficiente allocare un buffer che abbia come identificatore proprio la parola `mandatoryQueue`. Dopodiché, sarà sufficiente allocare una inport ad esso associata e scrivere il codice di gestione delle richieste provenienti dal processore:

```
MessageBuffer          mandatoryQueue,
                       ordered = "false",
                       abstract_chip_ptr = "true";

in_port(mandatoryQueue_in, CacheMsg, mandatoryQueue, desc = "..") {
  if (mandatoryQueue_in.isReady()) {
    peek(mandatoryQueue_in, CacheMsg) {
      if (
           cacheMemory.isTagPresent(in_msg.Address) == false &&
           cacheMemory.cacheAvail(in_msg.Address) == false
       ) {
        // make room for the block
        trigger(
           Event:Replacement,
           cacheMemory.cacheProbe(in_msg.Address)
        );
      } else {
        trigger(mandatory_request_type_to_event(in_msg.Type),
                in_msg.Address);
      }
    }
  }
}
```

Il costrutto `peek` serve per leggere il messaggio che è in testa ad un buffer; il messaggio viene semplicemente letto, non ne viene fatto il pop. È necessario verificare preventivamente che il buffer non sia vuoto utilizzando la funzione `isReady()` della `in_port`. Le funzioni `isTagPresent()` e `cacheAvailable()` della classe `CacheMemory` servono per verificare se un indirizzo è presente in cache (la prima) oppure se c'è posto per il nuovo blocco (la seconda) da caricarsi dalla memoria. La funzione `cacheProbe()` della classe `CacheMemory` implementa la politica LRU. Notare che la variabile `in_msg` è una variabile locale al blocco `peek` non definita dal programmatore, che rappresenta il messaggio contenuto in testa al buffer.

2.5.5 Le Action

Le action rappresentano delle sequenze di istruzioni che hanno un significato particolare. Esse contengono il codice SLICC che implementa le funzionalità della macchina a stati. Ad esempio, si può scrivere una action che invia il messaggio di lettura verso la memoria, oppure quella che alloca o che dealloca un blocco in cache.

Sintatticamente, una action è come una funzione dichiarata tramite il costrutto `action`, che prende tre parametri: il primo è l'identificativo della action stessa, il secondo è una stringa usata da Ruby quando crea le tabelle delle transizioni, ed il terzo è una descrizione di quello che fa la action:

```
action(a_issueRequest, "a", desc = "Issue a request") {
  enqueue(
      requestNetwork_out, RequestMsg, latency = "ISSUE_LATENCY"
  ) {
    out_msg.Address := address;
    out_msg.Type := CoherenceRequestType:GETX;
    out_msg.Requestor := machineID;
    out_msg.Destination.add(map_Address_to_Directory(address));
    out_msg.MessageSize := MessageSizeType:Control;
  }
}

action(
     e_sendData, "e", desc = "Send data from cache to requestor"
) {
  peek(forwardRequestNetwork_in, RequestMsg) {
    enqueue(
     responseNetwork_out,
     ResponseMsg,
     latency = "CACHE_RESPONSE_LATENCY"
    ) {
      out_msg.Address := address;
      out_msg.Type := CoherenceResponseType:DATA;
      out_msg.Sender := machineID;
      out_msg.Destination.add(in_msg.Requestor);
```

```
        out_msg.DataBlk := cacheMemory[address].DataBlk;
        out_msg.MessageSize := MessageSizeType:Response_Data;
      }
    }
  }
```

Analogo al costrutto *peek* è il costrutto *enqueue*, che accoda un messaggio in un buffer associato ad una outport; il primo parametro è l'identificatore della outport, il secondo è dato col costrutto latency e rappresenta la latenza di uscita del messaggio dal buffer. I campi del messaggio vengono riempiti all'interno del costrutto enqueue stesso. Notare che la variabile machineID è una variabile locale a tutta la macchina a stati non definita dal programmatore, che rappresenta l'identificatore unico del nodo nel sistema, mentre la variabile out_msg è una variabile locale al blocco enqueue non definita dal programmatore, che rappresenta il messaggio che si vuole accodare nel buffer di uscita.

2.5.6 Le Transition

Le transizioni sono una *sequenza atomica di azioni*. Una transizione viene iniziata a seguito del trigger da una inport di un evento per un particolare indirizzo: se l'indirizzo relativo al blocco per cui è stato scatenato l'evento è memorizzato nelle strutture dati della macchina a stati, parte la corrispondente transizione relativa alla coppia `<STATO,EVENTO>`, altrimenti ne parte una relativa alla coppia `<NON_PRESENTE,EVENTO>` (ad esempio, lo stato `Invalid` per la cache può valere come `NON_PRESENTE`).

Sintatticamente, una transizione si dichiara col costrutto `transition`, che prende come primo parametro lo stato di partenza, come secondo l'evento, e come terzo lo stato finale in cui la macchina a stati dovrà portarsi (si può omettere se la transizione non comporta cambiamento di stato). All'interno del costrutto `transition` saranno presenti una o più `action` preventivamente definite. Se più coppie `<STATO,EVENTO>` eseguono la stessa sequenza di action, *ed hanno lo stesso stato finale*, è possibile specificare come primo e secondo parametro una lista di stati e di eventi, racchiusi tra parentesi graffe e separati da virgole. Notare che nel caso in cui si utilizzi questa notazione compatta a lista, la transizione vale *per ogni possibile coppia* `<STATO,EVENTO>` derivante dal *prodotto cartesiano* fra la lista degli stati di partenza e quella degli eventi.

```
transition(I, Store, IM) {
  v_allocateTBE;
  i_allocateL1CacheBlock;
  a_issueRequest;
  m_popMandatoryQueue;
}

transition(IM, Data, M) {
  u_writeDataToCache;
  s_store_hit;
  w_deallocateTBE;
  n_popResponseQueue;
}

transition(M, Fwd_GETX, I) {
  e_sendData;
  o_popForwardedRequestQueue;
}

transition(M, Replacement, MI) {
  v_allocateTBE;
  b_issuePUT;
  x_copyDataFromCacheToTBE;
  h_deallocateL1CacheBlock;
}
```

2.6 Parametri di Configurazione

Nel file *rubyconfig.default* sono inseriti i parametri di configurazione della cache simulata. Tali parametri vengono utilizzati in fase di compilazione e "cablati" staticamente nel codice sorgente. È possibile comunque cambiarli a run-time, ma solo in fase di inizializzazione del sistema; ad esempio se si utilizza Ruby come modulo di Simics, basta usare il comando:

```
ruby0.setparam NOMEPARAMETRO VALORE
```

Elencati di seguito si trovano alcuni parametri ed il loro significato:

L1_CACHE_ASSOC:	Associatività della cache L1
L1_CACHE_NUM_SETS_BITS:	logaritmo in base 2 del numero di set della cache L1
L2_CACHE_ASSOC:	Associatività di ciascun banco della cache L2
L2_CACHE_NUM_SETS_BITS:	logaritmo in base 2 del numero di set di ciascun banco della cache L2
g_MEMORY_SIZE_BYTES:	dimensione in byte della memoria principale
g_DATA_BLOCK_BYTES:	dimensione in byte di un blocco di cache
g_NUM_PROCESSORS:	numero complessivo di processori del sistema
g_NUM_L2_BANKS:	numero di banchi della cache L2
g_NUM_MEMORIES:	numero di memorie del sistema
g_PROCS_PER_CHIP:	numero di processori per chip
L1_RESPONSE_LATENCY:	latenza della cache L1
L2_RESPONSE_LATENCY:	latenza del controller di un banco della cache L2
L2_REQUEST_LATENCY:	latenza di un singolo banco della cache L2
NETWORK_LINK_LATENCY:	latenza di un link della rete di interconnessione

2.7 Il test

Durante questo lavoro di tesi, la fase di test è stata la parte più importante di tutto il processo. Partendo dal una versione di base, che rappresentava il comportamento della S-NUCA, è stato inizialmente progettato il protocollo MESI PS-NUCA (§ 4) integrandolo nella macchina a stati della PS-NUCA. Una volta terminata la fase di implementazione, è iniziata quella di testing: lo strumento utilizzato è quello fornito a corredo di Ruby (§ 2.4). Il processo di testing è stato di tipo incrementale: all'individuazione da parte del tester di una nuova situazione di errore, di qualunque tipo, seguiva la relativa correzione. La tipologia dei problemi che il tester ha evidenziato ha comportato molti cambiamenti sia nel protocollo che in alcune scelte progettuali che erano state fatte all'inizio. In ogni caso, l'idea di base è stata conservata: il protocollo di base è rimasto

inalterato nelle sue fasi principali, ed il caso “base”, cioè quello che si verifica più spesso, è stato mantenuto semplice e veloce; i casi particolari, le corse critiche che sono emerse e gli errori comuni, sono stati trattati in modo da non impattare il caso medio, ma dovevano comunque essere affrontati per poter garantire il corretto funzionamento del sistema.

Metodologicamente il processo di test è cominciato con un sistema a due soli processori, collocati su due lati opposti della PS-NUCA nella configurazione 1+1P. Una volta che questo ha dato successo, è stato aumentato il numero di processori fino ad una configurazione 2+2P, ossia con quattro processori posti due da un lato, due al lato opposto: questo ha evidenziato ulteriori corse critiche che non potevano mai verificarsi nel caso 1+1P. Infine è stato effettuato il test ad otto processori nella configurazione 4+4P. Per ognuna di tali configurazioni il test è stato effettuato variando ulteriormente l’associatività: si è partiti da banchi di L2 ad accesso diretto, associativi a due vie e infine associativi a quattro vie. Ognuna di queste configurazioni è stata testata per un numero differente di riferimenti generati: un milione, tre milioni, cinque milioni e dieci milioni di riferimenti. Alla fine di tutta la procedura, il tester ha dato successo in tutte le configurazioni considerate e per ogni numero di riferimenti.

Capitolo 3

CMP PS-NUCA

In questo capitolo verrà descritto il sistema preso in considerazione. Si tratta di un sistema CMP a numero di processori generico, in cui la gerarchia di memoria comporta la presenza di una cache di primo livello (L1Cache o L1$) privata di ciascun processore e di una di secondo livello (L2Cache o L2$) organizzata secondo il paradigma S-NUCA, suddivisa in banchi privati associati a ciascun processore. E' inoltre presente una ulteriore cache (DirectoryCache o Dir$) che memorizza esclusivamente i blocchi di directory contenenti le informazioni relative alla Directory situata nella memoria principale. L'infrastruttura di comunicazione è una Network-on-Chip (NoC).

3.1 L1Cache

Ciascun processore del sistema è dotato di una cache di primo livello privata. La cache di primo livello è suddivisa in due parti: la InstructionCache (I$) e la DataCache (D$). Quando arriva una richiesta dal processore per un blocco di memoria, se si tratta di fetch di un'istruzione, il controllore della cache va a vedere nella I$, altrimenti controlla nella D$. In caso di hit, si restituisce immediatamente il controllo al processore. In caso di miss, invece, la richiesta parte verso la L2$, secondo le modalità che dipendono dal tipo di richiesta che è stata ricevuta dalla CPU.

3.1.1 Struttura di una linea della L1Cache

```
structure(
  Entry, desc = "L1-Cache Entry", interface = "AbstractCacheEntry"
) {
  State         CacheState,     desc = "Cache Block State";
  DataBlock     DataBlk,        desc = "Data Block";
}
```

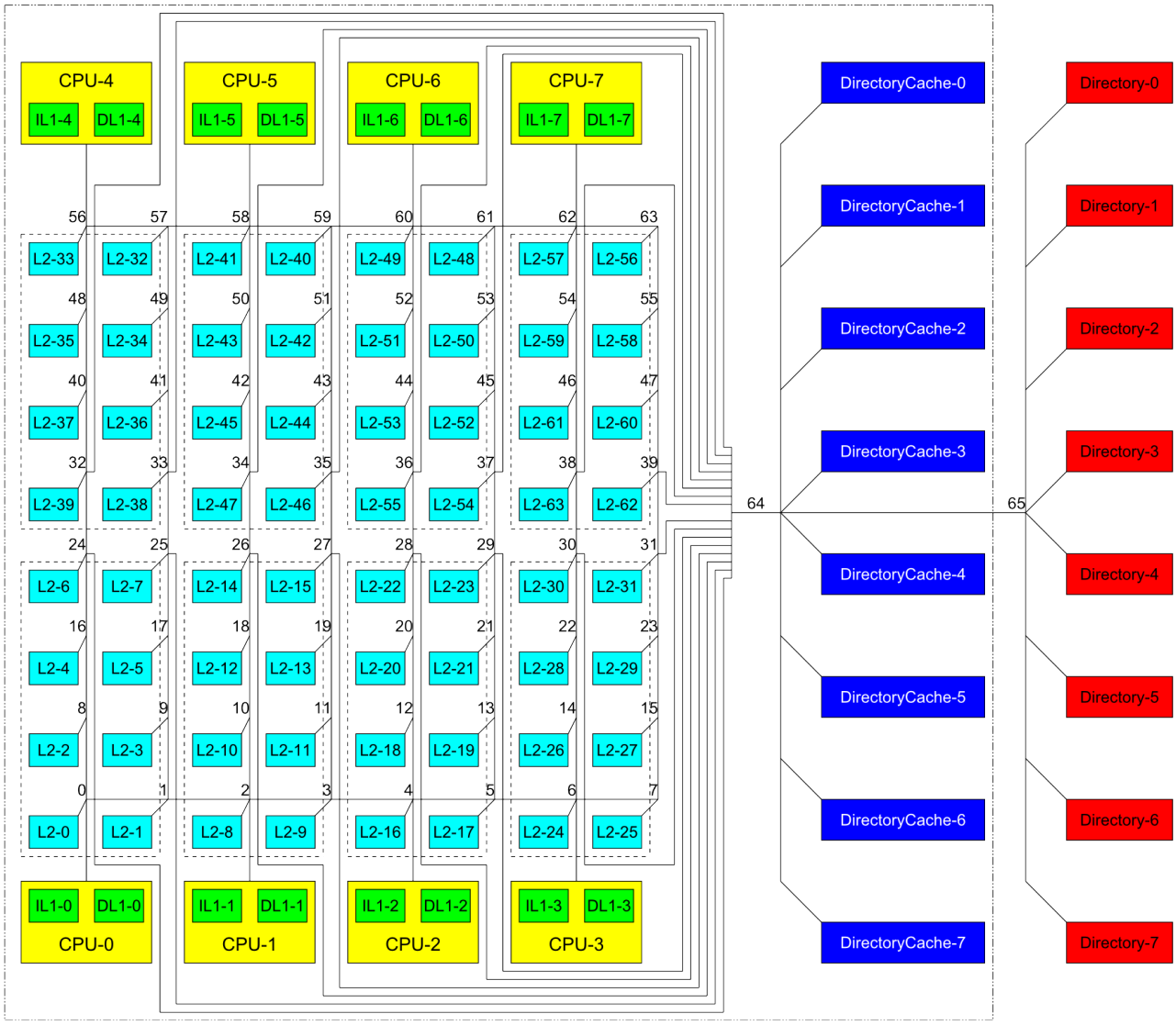

Figura 5 Sistema CMP PS-NUCA

Come si può notare, si necessita solamente dell'informazione inerente lo stato del blocco (`CacheState`), oltre ovviamente al blocco di memoria stesso (`DataBlk`).

In alcuni stati transienti, per ciascun blocco viene allocato anche un insieme di registri veloci (MHSR – Miss Handling Status Registers) che in Ruby vengono implementati attraverso le TBE (Transaction Buffer Entry), destinate a contenere informazioni relative alla gestione di situazioni particolari.

3.1.2 Struttura di una TBE della L1Cache

```
structure(TBE, desc = "Transaction Buffer Entry") {
    Address     Address,   desc = "Physical Address";
    State       TBEState,  desc = "Transient State";
}
```

Nella TBE si necessita invece dell'indirizzo fisico associato al blocco, nonché l'informazione inerente allo stato transiente in cui si trova il blocco stesso.

3.2 L2Cache

La cache di secondo livello è costituita da un'insieme banchi associati in modo privato a ciascun processore del sistema. Essa è organizzata secondo il paradigma S-NUCA. In figura 3-1 è riportato un sistema ad 8 processori, disposti su due lati della S-NUCA. La cache è organizzata come una matrice di banchi.

3.2.1 Struttura di una linea della L2Cache

```
structure(
  Entry, desc = "L2-Cache Entry", interface = "AbstractCacheEntry"
){
  State         CacheState,     desc = "Cache Block State";
  DataBlock     DataBlk,        desc = "Data Block";
  bool          L1Invalid,      desc = "L1-Cache Replaced";
}
```

La linea di L2Cache, necessita di un'informazione addizionale rispetto alla linea della L1Cache: `L1Invalid`. Tale flag indica la presenza o meno della copia del blocco di memoria nella L1$ associata al banco della L2$ in cui tale copia risiede, ma solo nel caso in cui la copia non sia condivisa con altri banchi di L2$ associati ad altri processori.

Anche nella L2$ è possibile allocare TBE per un blocco di cache; la struttura della TBE della L2$ è la seguente (non tutti i campi vengono usati contemporaneamente, per cui in ciascuna particolare situazione è significativo solo il valore dei campi effettivamente inizializzati).

3.2.2 Struttura di una TBE della Cache L2

```
structure(TBE, desc = "Transaction Buffer Entry") {
  Address       Address,  desc      =      "Physical      Address";
  State         TBEState, desc      =      "Transient       State";
  int           NumPendingAcks,
     desc     =     "Number     of     Pending     Acknowledgement";
  NetDest       ForwardGetS_IDs,
     desc  =  "Set  of  the  processors  to  forward  the  block";
  MachineID     ForwardGetX_ID,
     desc  =  "ID  of  the  processor  to  forward  the  block";
  bool          ForwardGetX_ID_present,
     desc = "ID of the processor to forward the block is present?";
  int           ForwardGetX_AckCount,
     desc = "Number of Acknowledgements the forwarded GetX needs";

}
```

Oltre alle informazioni viste in precedenza per la TBE della L1$, in quella relativa alla L2$ sono presenti informazioni addizionali necessarie al corretto funzionamento del protocollo di coerenza.

In particolare:

- `NumPendingAcks`: Numero delle accettazioni pendenti di ricevuta richiesta di invalidazione.
- `ForwardGetS_IDs`: Lista degli identificatori di banchi di L2$ a cui inviare il blocco in modo condiviso non appena possibile.
- `ForwardGetX_ID`: Identificatore del banco di L2$ a cui inviare il blocco in modo non condiviso non appena possibile.
- `ForwardGetX_AckCount`: Numero delle accettazioni di ricevuta invalidazione che il banco di L2$ deve attendere dopo aver ricevuto il blocco in modo non condiviso.

3.3 DirectoryCache

La DirectoryCache è costituita da banchi associati ciascuno ad ogni banco di Directory a sua volta associato ad ogni banco di memoria principale.

3.3.1 Informazioni memorizzate in una linea di DirectoryCache

```
structure(
  Entry,
  desc = "Directory Cache Entry",
  interface = "AbstractCacheEntry"
) {
    State       DirectoryState, desc = "Directory State";
    NetDest     Sharers,        desc = "Sharers List";
    bool        DirOwner,       desc = "Directory Owner";
    MachineID   ProcOwner,      desc = "Processor Owner";
}
```

A differenza di una cache di primo e/o di secondo livello, una linea di Dir$ memorizza solo ed esclusivamente il blocco di directory con le informazioni necessarie alla gestione del protocollo di coerenza.

In particolare:

- `DirectoryState`: Stato del blocco di Directory.
- `Sharers`: Lista degli identificatori dei banchi di L2$ che condividono un blocco di memoria.
- `DirOwner`: Indica se la Directory è “proprietaria” del blocco di memoria.
- `ProcOwner`: Identificatore del banco di L2$ “proprietario” del blocco di memoria.

A differenza delle cache di primo e di secondo livello, la Dir$ non necessita di alcuna TBE.

3.4 Politica di associazione dell'indirizzo fisico al banco di L2Cache privata.

Ciascun blocco di memoria è associato, sulla base del suo indirizzo fisico e del processore (L1$) che ne fa richiesta, in un preciso banco della S-NUCA. In particolare, supposto che l'indirizzo fisico del blocco sia di N bit, il banco della L2$ è individuato in base alla seguente politica.

Sia:

- A l'indirizzo fisico del blocco di memoria.
- $A = a_{N-1}a_{N-2}\cdots a_2 a_1 a_0 \qquad a_i \in \{0,1\};\ \ 0 \le i \le N-1;\ \ i, N \in \mathbb{N}$
- BS la dimensione in byte del blocco di memoria.
- NC il numero totale dei banchi della L2Cache.
- NP Il numero totale dei processori (L1Caches).
- $M = \log_2(NP)$
- P I numero del processore (L1Cache) di riferimento.
- $P = p_{M-1}\,p_{M-2}\cdots p_2\,p_1\,p_0 \qquad p_j \in \{0,1\};\ \ 0 \le j \le M-1;\ \ j, M \in \mathbb{N}$

Si considerino i parametri:

- $B = \log_2(BS)$ il numero di bit necessari ad indirizzare il byte nel blocco.
- $S = \log_2(NC/NP)$ il numero di bit necessari ad indirizzare il banco relativo ad ogni processore.

Il banco relativo ad ogni processore è dato dagli S bit meno significativi del campo `index` dell'indirizzo:

$$a_{N-1}\cdots a_{B+S}\,|\,a_{B+S-1}\cdots a_B\,|\,a_{B-1}\cdots a_0$$

Il banco effettivamente indirizzato invece è dato dalla concatenazione dei bit che compongono:

$$p_{M-1}\cdots p_0\,|\,a_{B+S-1}\cdots a_B$$

L'associazione inversa invece è indipendente dall'indirizzo fisico del blocco di memoria in quanto un banco di L2$, essendo privato, determina univocamente il processore (L1$) come sopra esposto.

3.5 Politica di associazione dell'indirizzo fisico al banco di Memoria

Ciascun blocco di memoria è associato, sulla base del suo indirizzo fisico, ad un solo banco di DirectoryCache, Directory oppure memoria principale. In particolare, supposto che l'indirizzo fisico del blocco sia di N bit, il banco della L2Cache è individuato in base alla seguente politica.

Sia:

- A l'indirizzo fisico del blocco di memoria.
- $A = a_{N-1} a_{N-2} \cdots a_2 a_1 a_0 \qquad a_i \in \{0,1\};\ \ 0 \le i \le N-1;\ \ i, N \in \mathbb{N}$
- BS la dimensione in byte del blocco di memoria.
- ND il numero totale dei banchi di Dir$, Directory oppure memoria principale.

Si considerino i parametri:

- $B = \log_2(BS)$ il numero di bit necessari ad indirizzare il byte nel blocco.
- $S = \log_2(ND)$ il numero di bit necessari ad indirizzare il banco di Dir$, Directory oppure memoria principale.

Il banco di Dir$, Directory oppure memoria principale è dato dagli S bit meno significativi del campo `index` dell'indirizzo:

$$a_{N-1} \cdots a_{B+S} | a_{B+S-1} \cdots a_B | a_{B-1} \cdots a_0$$

3.6 La NoC

L'infrastruttura di comunicazione, come si può notare dalla figura 3-1, è una NoC costituita da tanti switch quanti sono i banchi di della NUCA. Gli switch sono interconnessi con ciascun banco della NUCA attraverso dei link la cui latenza di attraversamento si assume essere pari ad 1 ciclo di clock.

La topologia della rete è di tipo partial 2D mesh network, in cui non c'è una maglia completa di link orizzontali e verticali, ma solo un insieme di collegamenti verticali (con latenza pari ad 1 ciclo di clock), più tanti collegamenti orizzontali quanti sono i lati della NUCA su cui i processori sono

connessi (con latenza pari a 2 cicli di clock).

La politica di routing della rete è di tipo X-Y: un messaggio inviato da una L1Cache che deve raggiungere un dato banco della NUCA, percorrerà sempre un path caratterizzato, se esiste, da un percorso orizzontale fino al raggiungimento del percorso verticale relativo al banco di interesse. Caso particolare è quello in cui la L1$ mittente è direttamente connessa allo switch del banco cui il messaggio è destinato: in questo caso lo switch instraderà il messaggio direttamente sulla porta di uscita che è connessa al banco (cioè, sulla colonna). Resta inteso che un messaggio inviato, in risposta, da un banco della NUCA percorre sempre il path verticale e, successivamente se necessario, quello orizzontale.

Le DirectoryCaches invece sono tutte connesse ad uno switch, situato “al di fuori” della mesh network che collega la NUCA ai vari processori (L1$), con collegamenti aventi latenza pari ad 1 ciclo di clock. Tale switch è connesso alla mesh network per mezzo di 16 collegamenti, aventi latenza pari ad un ciclo di clock, con altrettanti switch. Inoltre lo switch che collega le Dir$ alla mesh network ha un ulteriore collegamento verso le Directories situate in memoria principale all'esterno del Chip.

3.7 Protocollo di coerenza

Il protocollo di coerenza di base dal quale è partita l’implementazione della PS-NUCA è un protocollo MESI a directory. Le informazioni di directory sono contenute nei blocchi di Directory in memoria principale, nei blocchi della DirectoryCache oppure nella linea di cache (L1$ oppure L2$) che contiene la copia del blocco di memoria.

Un protocollo di coerenza di tipo MESI si basa sulla distinzione di quattro possibili stati in cui può trovarsi un blocco di memoria nella L1Cache o in un qualsiasi banco della L2Cache privata:

- **Modified:** il blocco di cache non è condiviso, è stato soggetto ad operazioni di scrittura e/o lettura e non è coerente con la copia in memoria principale.

- **Exclusive**: il blocco di cache non è condiviso, è stato soggetto a sole operazioni di lettura ed è coerente con la copia in memoria principale.

- **Shared**: il blocco di cache è condiviso e tale copia è la stessa su tutte le L1$ e nei banchi delle L2$ ed è coerente con la copia in memoria principale.

- **Invalid**: il blocco di cache non è valido ossia la copia del blocco di memoria non è presente nella cache.

Quando una copia di un blocco di memoria è considerato non condiviso (si trova nello stato Modified o Exclusive), in caso di rimpiazzamento è necessario notificare tale evento alla cache di livello inferiore o eventualmente alla Directory. Nel caso in cui il blocco di cache sia nello stato Exclusive basta un semplice messaggio di notifica, mentre nel caso il blocco sia nello stato Modified è necessario inviare la copia del blocco modificato per aggiornarne la cache di livello inferiore o addirittura la memoria principale. Nel caso di rimpiazzamento di un blocco di cache nello stato Shared non è necessario che alcuna cache invii messaggi di notifica al livello inferiore o addirittura alla Dir$ (rimpiazzamento silente).

Se il rimpiazzamento avviene in un banco di L2$ è necessario notificare tale evento alla relativa L1$, la quale dovrà procedere ad una eventuale all'invalidazione del blocco, nel caso sia quest'ultimo sia condiviso, oppure effettuare un eventuale rimpiazzamento “forzato” nel caso in cui il blocco si venisse a trovare nello stato Modified e Exclusive (principio di inclusione).

Capitolo 4
Il Protocollo di Coerenza

In questo capitolo viene descritto in maniera dettagliata il protocollo di coerenza MESI a directory per il sistema CMP PS-NUCA. Per semplicità di trattazione si indicherà con L1-X, l'insieme dei banchi I$ e D$ della L1$ associata al processore X; con L2-X, l'insieme dei banchi della L2$ associati come privati sempre al processore X; con N il banco di Dir$ e/o Directory associato all'indirizzo fisico del blocco di directory e/o di memoria per cui vengono effettuate le varie richieste.

4.1 Directory e Directory Cache

La Dir$ è una cache per i blocchi di directory che dalla memoria principale transitano on-chip e viceversa. In tali blocchi vengono memorizzate appunto le informazioni di directory necessarie per il corretto funzionamento del protocollo. Di conseguenza la Dir$ è soggetta a miss ed a rimpiazzamenti dovuti al fatto che, essendo di dimensioni ridotte rispetto alla Directory situata in memoria principale, non è in grado di contenere contemporaneamente tutti blocchi.

In Figura 6 vengono illustrate le interazioni tra Directory e Dir$ per la gestione delle miss e dei rimpiazzamenti scatenati appunto nella Dir$.

Si supponga che:

- un banco di L2-X debba effettuare una richiesta (L2-X_REQUEST) alla Dir$ per un blocco di memoria il cui blocco di directory non è presente nella Dir$ stessa

- il blocco di directory, relativo al blocco di memoria richiesto, si trovi inizialmente nello stato R ad indicare che esso non è presente nella Dir$ stessa.

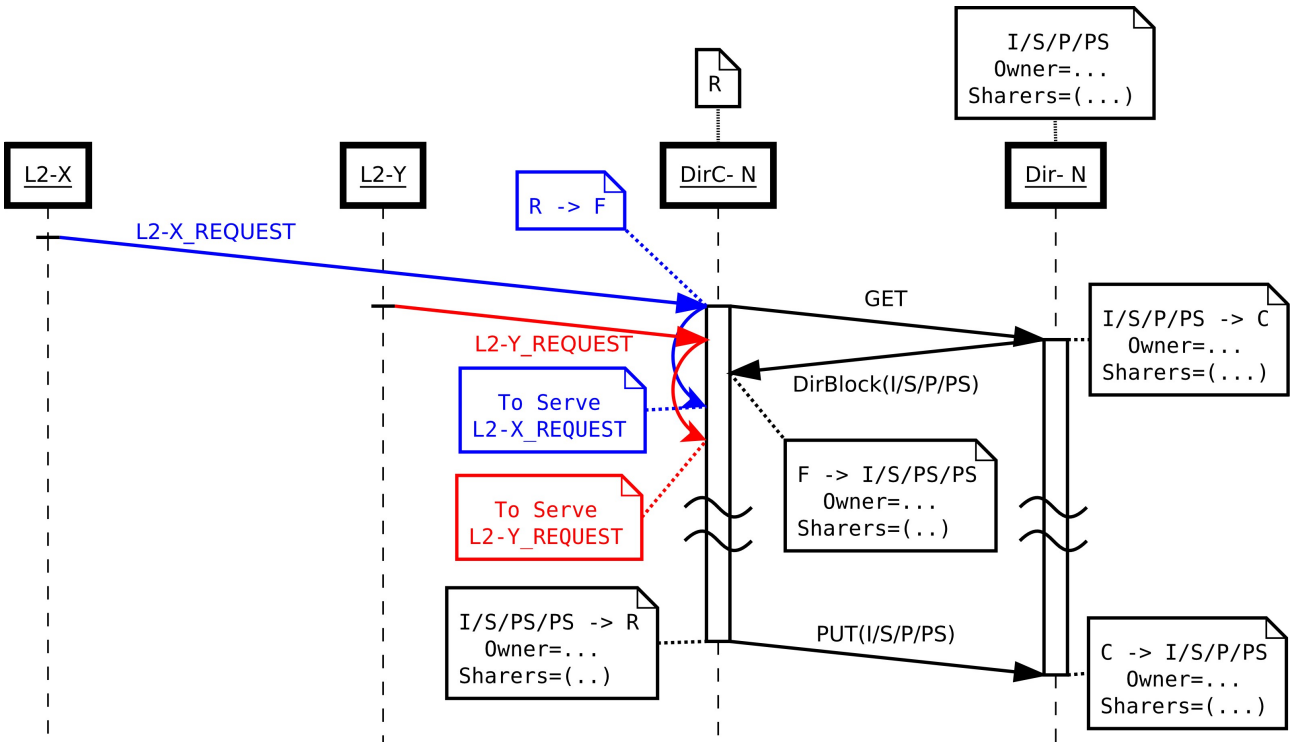

Figura 6 Miss e rimpiazzamento della DirectoryCache

Nell'istante in cui la L2-X_REQUEST viene schedulata dalla coda delle richieste della Dir$, quest'ultima invia una richiesta di blocco di directory (GET) verso la Directory utilizzando una *virtual network* in grado di garantire l'ordinamento dei messaggi. Siccome la L2-X_REQUEST non può essere servita per assenza del blocco di directory nella Dir$, essa viene rimessa in coda in modo da poterla rischedulare quando il blocco di directory verrà effettivamente trasferito nella Dir$. Infine, per il blocco in questione, la Dir$ transisce nello stato F indipendentemente dal tipo di richiesta arrivata. Finchè il blocco di directory non arriva alla Dir$, qualsiasi altra richiesta proveniente da qualsiasi altro banco di L2$ (Es. L2-Y_REQUEST) per il blocco di memoria in questione viene rimessa in coda in quanto non è possibile gestirla.

Il blocco di directory memorizzato in memoria principale, potrà trovarsi in qualsiasi stato coerente con il protocollo e, non appena la richiesta di tale blocco proveniente dalla Dir$ viene schedulata dalla coda delle richieste della Directory, quest'ultima provvederà ad inviare il blocco in oggetto alla Dir$, transendo nello stato C.

Nell'instante il blocco di directory arriva alla Dir$, quest'ultima lo memorizza e transisce dallo stato F allo stato indicato nel blocco stesso. Successivamente tutte le richieste potranno essere servite.

Essendo la Dir$ a capacità limitata, può capitare che un blocco di directory sia selezionato come vittima di un rimpiazzamento da parte dell'algoritmo LRU implementato nella Dir$ stessa. Quando ciò accade, la Dir$ invia un messaggio (PUT) contente il blocco di directory da rimpiazzare alla Directory utilizzando la *virtual network* ordinata e transisce nello stato R, permettendo di fatto il rimpiazzamento stesso.

Per semplicità, d'ora in poi, si supporrà che qualsiasi blocco di directory sia sempre presente nella Dir$ nell'istante in cui le varie richieste vengo schedulate, evitando di fatto il ripetersi del trattamento delle miss nella Dir$ stessa.

4.2 L1 Hit

Verrà ora discusso come si comporta il protocollo quando un processore abbia bisogno di effettuare un'operazione su un blocco di memoria che è effettivamente presente nella L1Cache (L1 Hit).

4.2.1 Load oppure iFetch

Nello scenario in cui un generico processore abbia bisogno di effettuare un'operazione di Load oppure iFetch (operazioni di lettura) su un blocco di memoria che è effettivamente presente nella L1Cache, indipendentemente dalla stato in cui il blocco si trova, nella L1$ si scatenerà una Load Hit che permetterà al processore di portare a termine l'operazione senza avere nessuna transizione di stato per il blocco in cache.

4.2.2 Store

Nello scenario in cui, invece, l'operazione sia di Store, la Store Hit, che permetterà al processore di portare a termine l'operazione, verrà scatenata se e solo se il blocco in cache si trova nello stato M oppure E. Come conseguenza, la L1$ farà transire il blocco in cache nello stato M per effetto dell'operazione di scrittura. Lo scenario in cui un processore debba effettuare una operazione di Store su un blocco di memoria, presente nella L1$, ma che si trovi nello stato S verrà discusso in seguito in quanto, pur essendo in presenza di una Hit, essa viene trattata dal protocollo in maniera differente.

4.2.3 Rimpiazzamenti in L1Cache

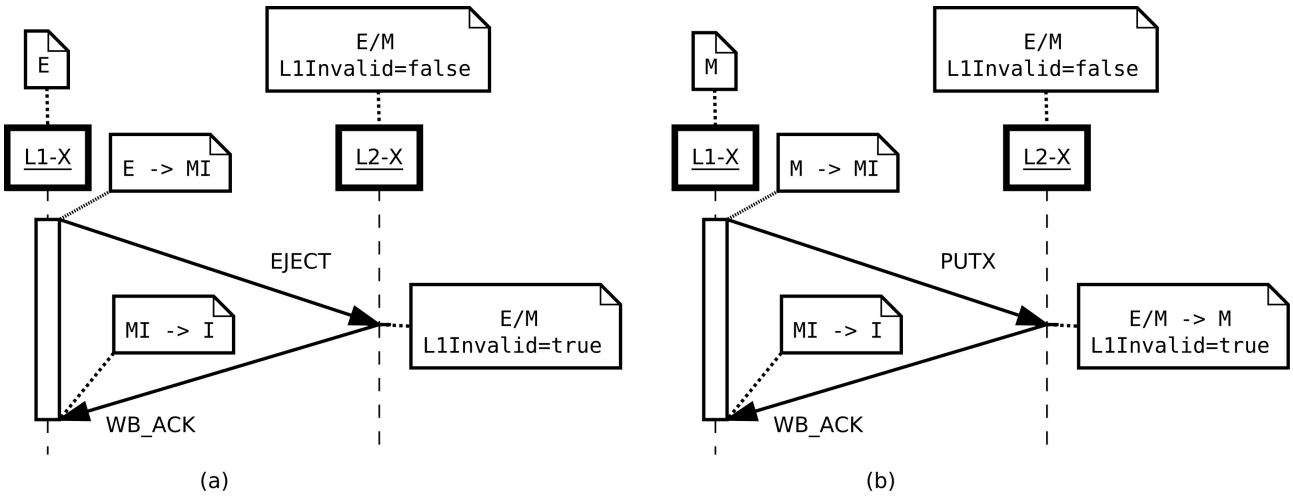

Figura 7 Rimpiazzamenti di blocchi non condivisi in L1Cache

Si supponga che in un determinato istante, un blocco di memoria la cui copia è presente in una L1Cache, sia selezionato come vittima di un rimpiazzamento dall'algoritmo LRU implementato nella L1$. Se tale blocco è condiviso, ossia si trova nello stato S, il rimpiazzamento avviene in modo silente, ossia non si comunica nulla al relativo banco della L2Cache, facendo transire il blocco nello stato I. In caso contrario il protocollo di coerenza si comporta come illustrato nella Figura 7.

Il blocco di cache in un banco della L2-X contenente la copia del blocco di memoria da rimpiazzare in L1-X può trovarsi, indifferentemente, nello stato E oppure nello stato M; con il flag `L1Invalid` impostato a `false`.

Il diagramma *a* illustra lo scenario in cui il blocco di memoria in L1-X si trova nello stato E, ossia non è stato modificato rispetto alla copia contenuta nel relativo banco in L2-X. Quindi la L1-X invia alla L2-X un messaggio di tipo `EJECT` che non contiene il blocco di memoria, finendo poi per transire nello stato MI: Modified to Invalid. La L2-X, vedendosi recapitare una `EJECT`, non deve far altro che impostare il flag `L1Invalid` a `true` e rispondere alla L1-X con un messaggio `WB_ACK` (Write-Back Acknowledgement) senza transire di stato. La L1-X infine, ricevuto il `WB_ACK`, transisce nello stato I.

Il diagramma *b*, invece, illustra lo scenario in cui il blocco di memoria in L1-X si trova nello stato M, quindi modificato rispetto alla copia contenuta nel relativo banco della L2-X. Adesso la L1-X deve inviare la copia modificata del blocco di memoria alla L2-X e lo fa incapsulando il blocco nel messaggio di tipo `PUTX`, andando a transire nello stato MI. La L2-X, ricevendo la `PUTX` che contiene un blocco modicato, lo memorizza, imposta a `true` il flag `L1Invalid`, invia il `WB_ACK` alla L1-X ed infine transisce nello stato M.

4.2.4 Rimpiazzamenti in L2Cache

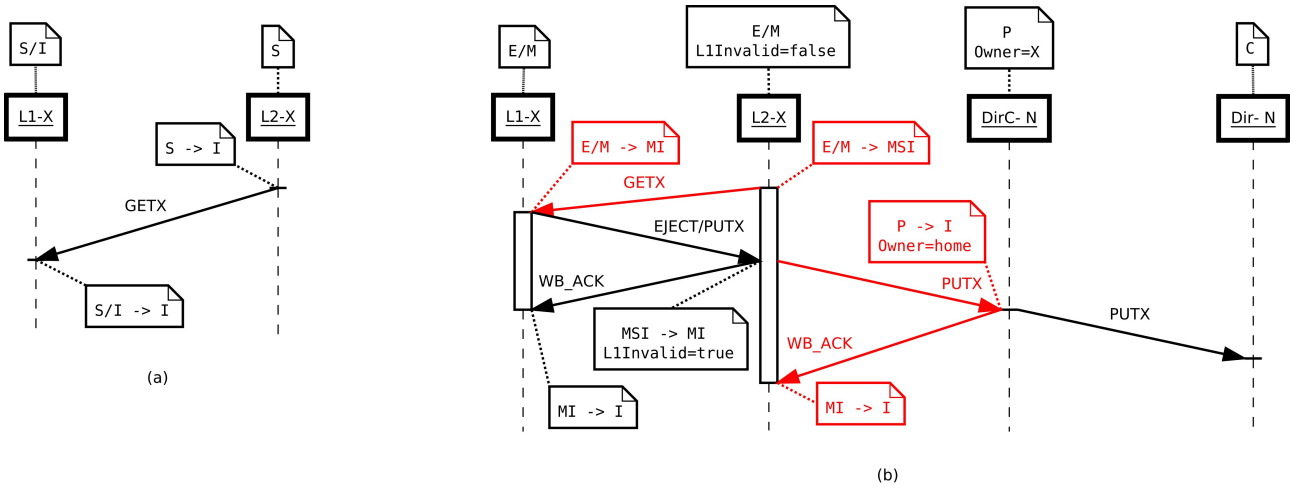

Figura 8 Rimpiazzamenti di blocchi in L2Cache

Si supponga che in un determinato istante, un blocco di memoria, la cui copia è presente sia in L1Cache, sia in un banco della L2Cache, venga selezionato come vittima di un rimpiazzamento dall'algoritmo LRU implementato nella L2$.

La Figura 8 illustra due scenari in cui il blocco di memoria sia (a) o non sia (b) condiviso.

Il diagramma *a* illustra che, quando si scatena un rimpiazzamento in un banco della L2-X per un blocco di memoria condiviso, la L2-X invia un messaggio di tipo `GETX` verso la L1-X e transisce direttamente nello stato I, non comunicando affatto con la Directory (rimpiazzamento silente).

La ricezione del messaggio `GETX` provoca nella L1-X un rimpiazzamento “forzato” e, di fatto, la L1-X si comporta come descritto in §4.2.3: effettuando un rimpiazzamento silente ossia andando a transire direttamente nello stato I.

Si noti che il blocco di cache nella L1-X potrebbe a sua volta essere già stato rimpiazzato in modo silente, per cui bisogna prevedere l'arrivo del messaggio `GETX` anche quando il blocco di cache nella L1-X è nello stato I ed ignorare, di fatto, il messaggio.

Il diagramma *b* illustra che, quando si scatena un rimpiazzamento in un banco della L2-X per un blocco di memoria non condiviso, la L2-X invia comunque il messaggio `GETX` verso la L1-X, che adesso non si invalida direttamente, ma transisce nello stato MSI in quanto deve attendere l'effettivo rimpiazzamento “forzato” da parte della L1-X.

La L1-X infatti, si comporta come descritto in §4.2.3: all'arrivo della `GETX` invia verso la L2-X il messaggio `EJECT`, se il blocco di cache si trova nello stato E; oppure invia verso la L2-X la copia del blocco di memoria modificato, incapsulandolo nel messaggio `PUTX`, se il blocco di cache si trova nello stato M; andando poi a transire nello stato MI.

La L2-X, nel caso in cui dovesse ricevere la `PUTX` contenente il blocco di memoria modificato, lo memorizza; ed in ogni caso, anche quando avesse ricevuto il messaggio `EJECT`, lo invia attraverso una `PUTX`, contenente anche l'identificatore del banco di L2-X che sta rimpiazzando, alla DirectoryCache; imposta il flag `L1Invalid` a `true`; invia il messaggio `WB_ACK` alla L1-X; ed infine transisce nello stato MI.

La Dir$ si trova nello stato P ed ha il campo `Owner` (che sintetizza, per semplicità, i due campi `DirOwner` e `ProcOwner` presenti effettivamente nel blocco di directory) impostato con l'identificatore del banco della L2$ che è l'effettivo “proprietario” del blocco (per semplicità X). Non appena riceve il blocco di memoria incapsulato nella `PUTX`, la Dir$ deve, per forza di cose, inoltrarlo alla Directory, in quanto non è in grado di memorizzarlo e per far ciò utilizza la *virtual network* ordinata. Inoltre invia, utilizzando sempre la *virtual network* ordinata, il messaggio `WB_ACK` al banco di L2-X che ha rimpiazzato (l'identificatore è contenuto nel messaggio `PUTX`); imposta il campo `Owner` ad `home`, diventando l'effettiva “proprietaria” del blocco; transisce nello stato I.

La Directory, il cui blocco di directory, relativo al blocco di memoria da rimpiazzare, è contenuto nella Dir$, si trova nello stato C. Vedendosi arrivare un blocco di memoria contenuto nel messaggio `PUTX`, non fa altro che memorizzarlo in memoria principale, senza transire di stato.

La L2-X, all'arrivo del WB_ACK proveniente dalla DiectoryCache, conclude il rimpiazzamento, facendo transire il blocco di cache nello stato I. La L1-X allo stesso modo, all'arrivo del WB_ACK proveniente dalla L2-X, conclude il rimpiazzamento facendo transire il blocco di cache nello stato I.

Si noti che, all'arrivo della GETX proveniente dalla L2-X, la L1-X potrebbe essere già in fase di rimpiazzamento ed aver già inviato EJECT/PUTX alla L2-X. In tal caso il protocollo gestisce lo scenario correttamente: bisogna solo prevedere l'arrivo di GETX nella L1-X quando essa è nello stato MI ed ignorare di fatto il messaggio.

4.3 L1 Miss ed L2 Hit

Verrà ora discusso come si comporta il protocollo quando un processore abbia bisogno di effettuare un'operazione su un blocco di memoria non presente nella L1Cache (L1 Miss), ma presente nel relativo banco della L2Cache associata (L2 Hit).

4.3.1 Load oppure iFetch

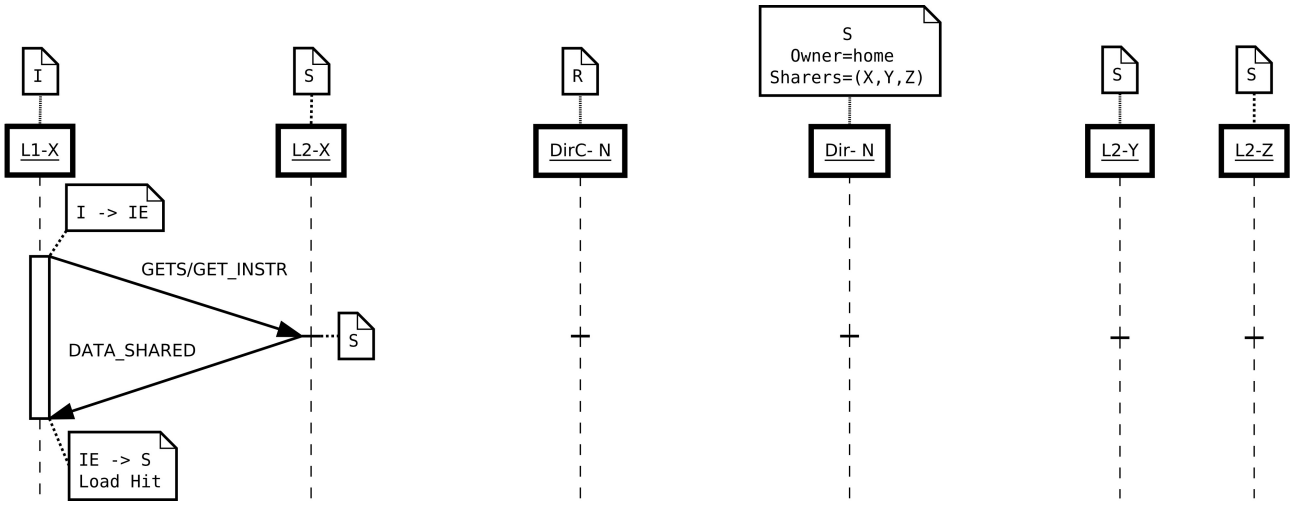

Figura 9 Load/iFetch con L1 Miss ed L2 Hit su blocco condiviso

La Figura 9 mostra lo scenario in cui un blocco di memoria è condiviso (ed è quindi presente, ad esempio, nei relativi banchi della L2$ associati ad i processori X, Y e Z) ed il processore X necessita di effettuare un'operazione di Load oppure di iFetch sul blocco di memoria in oggetto. I blocchi di cache nei vari banchi della L2$, essendo condivisi, dovranno trovarsi tutti nello stato S (salvo rimpiazzamenti silenti). Si può tranquillamente supporre che il blocco di directory relativo al

blocco di memoria in oggetto sia o non sia presente nella DirectoryCache in quanto non viene interessato. Il blocco di directory, a sua volta, deve trovarsi nello stato S con il campo `Owner` impostato ad `home` ad indicare che è effettivamente la Directory la “proprietaria” del blocco; inoltre il campo `Sharers` conterrà la lista degli gli identificatori dei banchi di L2$ che avevano in precedenza fatto richiesta per il blocco di memoria ed a cui il blocco è stato effettivamente inviato.

La miss nella L1-X a seguito della Load/iFetch impone a quest'ultima, trovandosi appunto nello stato I, di bloccare il processore e di confezionare una richiesta del blocco di memoria - `GETS (GET SHARED)` se si tratta si una Load oppure `GET_INSTR (GET INSTRUCTION)` se si tratta di una iFetch - da inviare al relativo banco della L2-X. La L1-X, a seguito di tale operazione, transisce nello stato IE.

Essendo il blocco di cache della L2-X nello stato S, la L2-X è in grado di soddisfare tale richesta, per cui, incapsula il blocco in questione in un mesaggio `DATA_SHARED` da inviare alla L1-X ad indicare appunto, che quest'ultima deve considerare il blocco come condiviso. La L2-X non transisce di stato.

La L1-X, vedendosi arrivare il messaggio `DATA_SHARED` contente la copia del blocco di memoria in precedenza richiesto, la memorizza, sblocca il processore permettendogli di effettuare la Load Hit ed infine transisce nello stato S.

La Figura 10 illustra lo scenario in cui il blocco di memoria è non è condiviso, ma presente solo in L2-X nello stato E oppure M. La Directory, per coerenza, deve essere nello stato P con il campo `Owner` contenente l'identificatore del banco della L2-X in cui il blocco risiede.

Come illustrato per lo scenario precedente, la L1 Miss provocherà nella L1-X, oltre al blocco del processore, l'invio della richiesta del blocco di memoria (`GETS/GET_INSTR`) verso la L2-X e la transione dallo stato I allo stato IE.

La L2-X, vedendosi arrivare la richiesta di cui sopra da parte della L1-X, è in grado di soddisfarla, per cui incapsula il blocco di memoria in un messaggio `DATA_EXCLUSIVE` da inviare come risposta alla L1-X ad indicare che ora, a differenza dello scenario precedente, il blocco può essere considerato esclusivo e non condiviso. La L2-X non transisce di stato, ma imposta il flag `L1Invalid` a `false` in quanto ora si ha la sicurezza la L1-X possiede anch'essa la copia del

blocco.

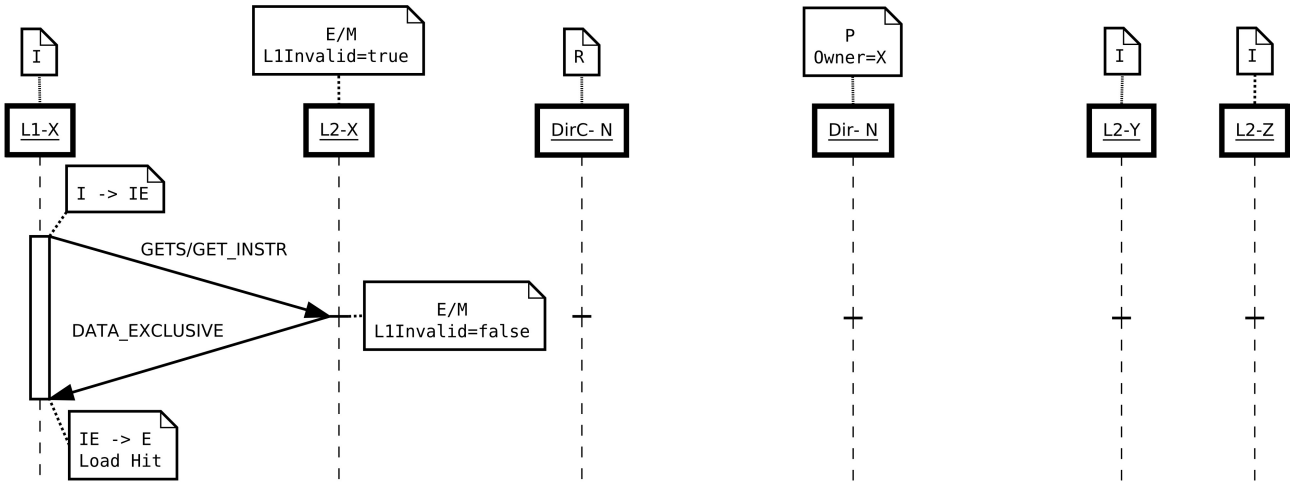

Figura 10 Load/iFetch con L1 Miss ed L2 Hit su blocco non condiviso

Infine la L1-X, vedendosi recapitare il messaggio `DATA_EXCLUSIVE`, contenente il blocco richiesto, lo memorizza, sblocca il processore permettendogli di effettuare la Load Hit e transisce nello stato E.

4.3.2 Store

La Figura 11 illustra uno scenario analogo al precedente, con la sola differenza che il processore X necessita di effettuare una operazione di Store.

La miss nella L1-X, impone a quest'ultima, dopo aver bloccato il processore, di inviare la richiesta per il blocco di memoria verso la L2-X in modo esclusivo e non condiviso: `GETX` (`GET EXCLUSIVE`). Di conseguenza transisce nello stato IM.

La L2-X si comporta in modo simile a quello descritto nello scenario precedente: inviando come risposta il messaggio `DATA_EXCLUSIVE` contenente il blocco di memoria, impostando il flag `L1Invalid` a `false`, ed andando infine a transire nello stato M.

Infine la L1-X, vedendosi recapitare il messaggio `DATA_EXCLUSIVE` contente il blocco richiesto, lo memorizza, sblocca il processore permmettendo la Store Hit e transisce nello Stato M, in quando ora possiede l'unica copia aggiornata del blocco.

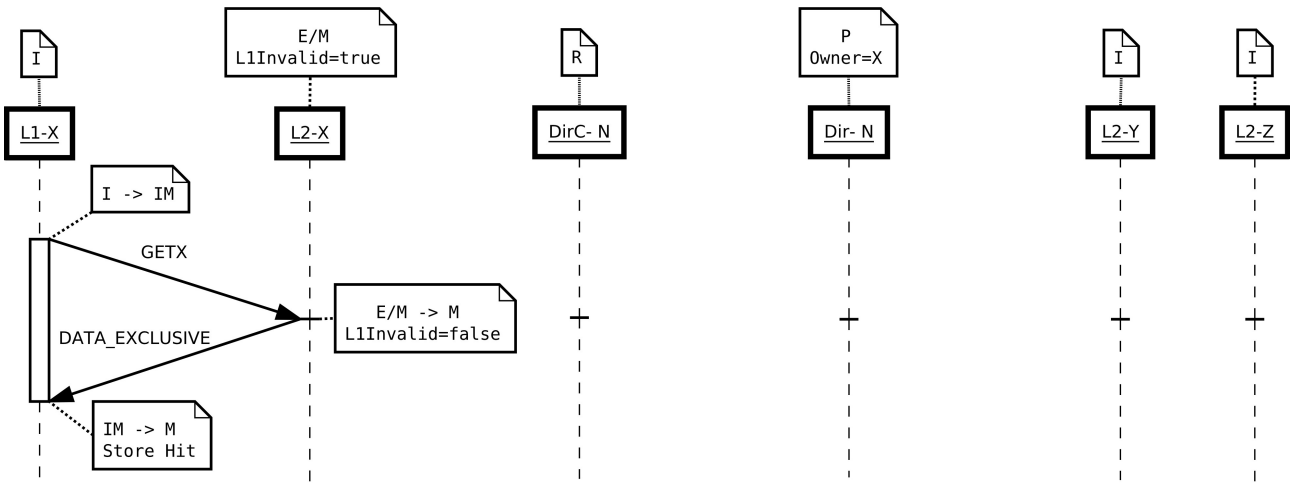

Figura 11 Store con L1 Miss ed L2 Hit su blocco non condiviso

Lo scenario in cui un processore debba effettuare una operazione di Store su un blocco di memoria, assente nella L1$ (L1 Miss) e presente nel relativo banco della L2$ (L2 Hit) nello stato S, verrà discusso in seguito in quanto, pur essendo in presenza di una Hit in L2$, essa viene trattata dal protocollo in maniera differente.

4.3.3 Rimpiazzamenti in L1Cache

La Figura 12 illustra lo scenario in cui un rimpiazzamento per un blocco di cache in L1-X venga scatenato quando tale blocco è nello stato IE: ossia dopo che la L1-X stessa abbia inviato la `GETS/GET_INSTR` alla L2-X, per un blocco di memoria condiviso, e prima che abbia ricevuto il `DATA_SHARED` da quest'ultima.

La L1-X si comporta come descritto in §4.3.1, ossia invia la richiesta del blocco di memoria per mezzo di `GETS/GET_INSTR` alla L2-X e transisce nello stato IE. La L2-X, alla ricezione della `GETS/GET_INSTR`, si comporta anch'essa come descritto in §4.3.1, inviando la copia del blocco di memoria per mezzo del messaggio `DATA_SHARED`.

Prima che arrivi il `DATA_SHARED`, contenente la copia del blocco di memoria richiesto, il blocco di cache in L1-X viene scelto dall'algoritmo LRU come vittima per un rimpiazzamento; imponendo allo stesso la trasizione nello stato IEI.

La L1-X, alla ricezione del `DATA_SHARED` contenente la copia del blocco di memoria, lo memorizza, sblocca il processore per permettere la Load Hit ed invia il messaggio `EJECT` alla L2$, iniziando, di fatto, la fase di rimpiazzamento, senza transire di stato.

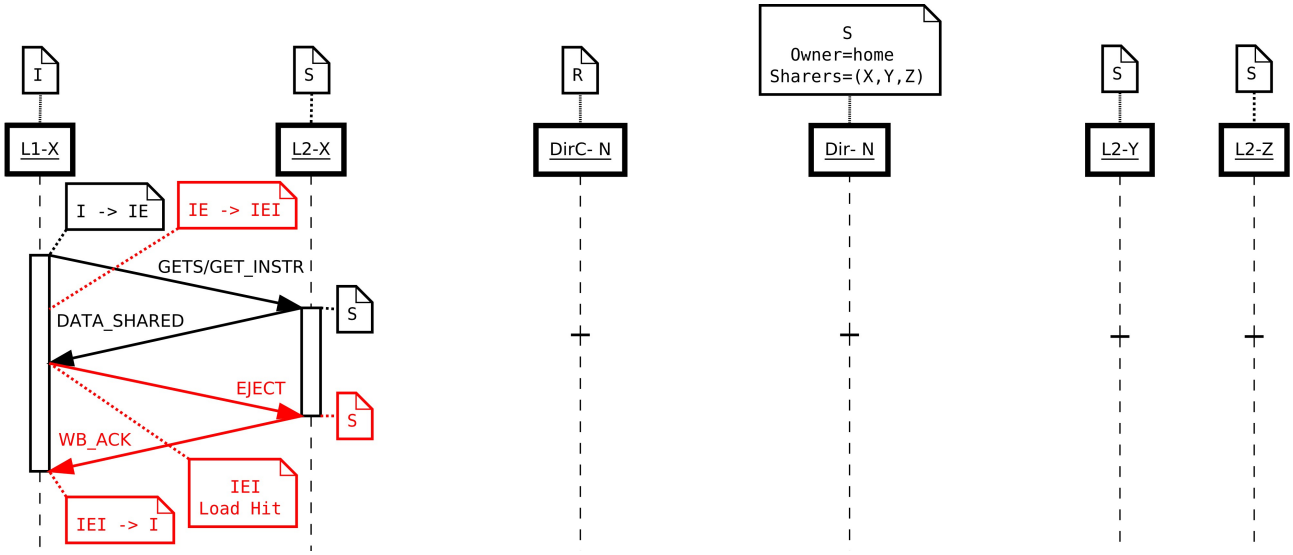

Figura 12 Rimpiazzamento post Load/iFetch per blocco di memoria condiviso

La L2-X, ricevuto il messaggio `EJECT`, senza transire di stato risponde alla L1-X inviandole il `WB_ACK` che, una volta ricevuto dalla L1-X stessa, la fa transire nello stato I concludendo di fatto il rimpiazzamento.

La Figura 13 illustra lo scenario precedente con la differenza che il blocco di memoria, la cui copia è memorizzata nella L2-X, adesso non è condiviso.

La L1-X si comporta come descritto in §4.3.1, ossia fa richiesta del blocco di memoria per mezzo di `GETS/GET_INSTR` alla L2-X e transisce nello stato IE.

La L2-X, alla ricezione della `GETS/GET_INSTR`, si comporta anch'essa come descritto in §4.3.1, inviando la copia del blocco di memoria per mezzo del messaggio `DATA_EXCLUSIVE` e impostando `L1Invalid` a `false`.

Prima che arrivi il `DATA_EXCLUSIVE`, contenente la copia del blocco di memoria richiesto, il blocco di cache in L1-X viene scelto dall'algoritmo LRU come vittima per un rimpiazzamento; imponendo allo stesso la trasizione nello stato IEI.

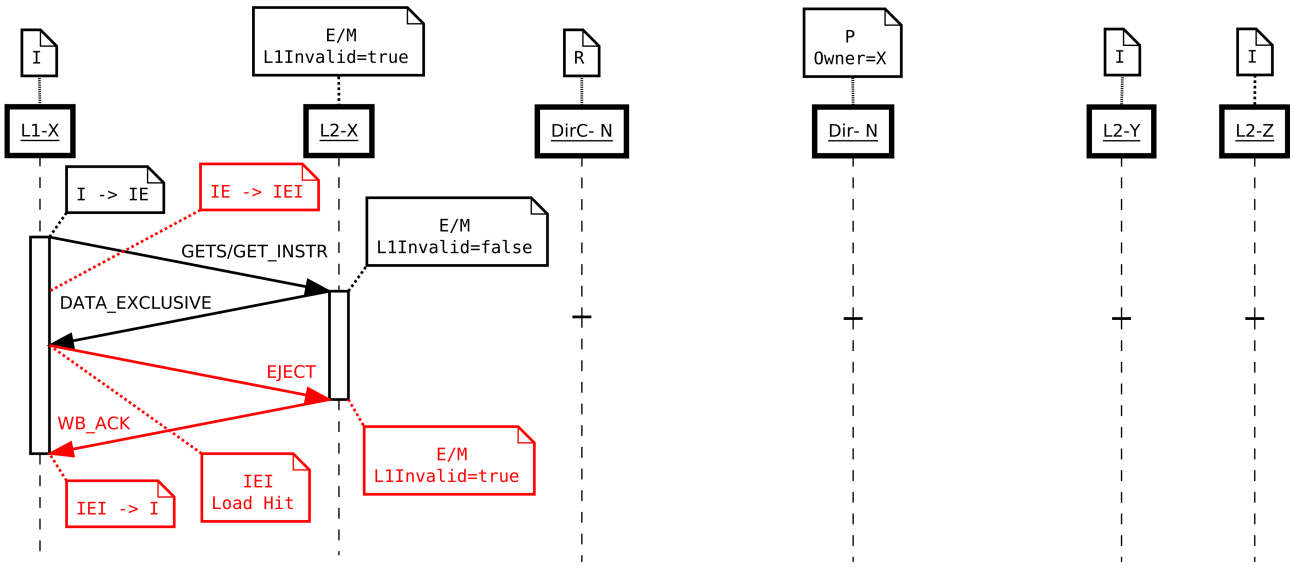

Figura 13 Rimpiazzamento post Load/iFetch per blocco di memoria non condiviso

La L1-X, alla ricezione del `DATA_EXCLUSIVE` contenente la copia del blocco di memoria, lo memorizza, sblocca il processore per permettere la Load Hit ed invia il messaggio `EJECT` alla L2$, iniziando, di fatto, la fase di rimpiazzamento, senza transire di stato.

La L2-X si comporta come descritto in §4.2.3, impostando `L1Invalid` a `true` e rispondendo alla L1-X con `WB_ACK` che, una volta ricevuto dalla L1-X, la fa transire nello stato I concludendo di fatto il rimpiazzamento.

La Figura 14 illustra lo scenario in cui un rimpiazzamento per un blocco di cache in L1-X venga scatenato quando tale blocco è nello stato IM: ossia dopo che la L1-X stessa abbia inviato la `GETX` alla L2$, per un blocco di memoria non condiviso, e prima che abbia ricevuto il `DATA_EXCLUSIVE` da quest'ultima.

La L1-X si comporta come descritto in §4.3.2, ossia fa richiesta del blocco di memoria per mezzo di `GETX` alla L2-X e transisce nello stato IM. La L2-X, alla ricezione della `GETX`, si comporta anch'essa come descritto in §4.3.2, inviando la copia del blocco di memoria per mezzo del messaggio `DATA_EXCLUSIVE`, impostando `L1Invalid` a `false`, ed andando a transire nello stato M.

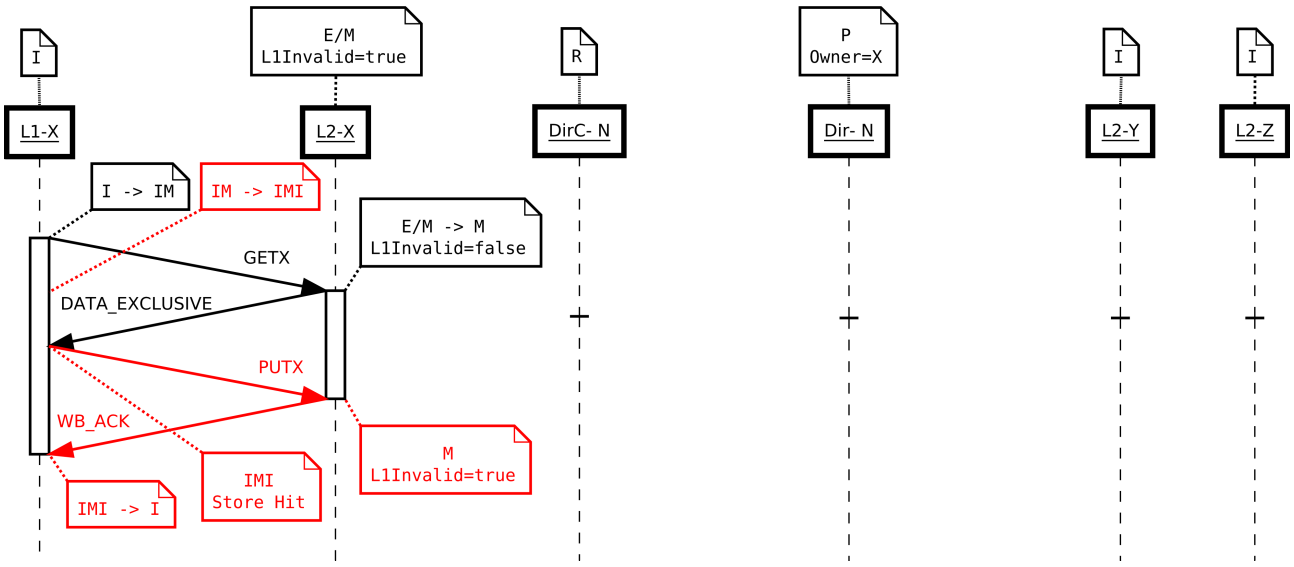

Figura 14 Rimpiazzamento post Store per blocco di memoria non condiviso

Prima che arrivi il `DATA_EXCLUSIVE`, contenente la copia del blocco di memoria richiesto, il blocco di cache in L1-X viene scelto dall'algoritmo LRU come vittima per un rimpiazzamento; imponendo allo stesso la trasizione nello stato IMI.

La L1-X, alla ricezione del `DATA_EXCLUSIVE` contenente la copia del blocco di memoria, lo memorizza, sblocca il processore per permettere la Store Hit ed invia la copia del blocco di memoria modificato alla L2-X per mezzo del messaggio `PUTX`, iniziando, di fatto, la fase di rimpiazzamento, senza transire di stato.

La L2-X si comporta come descritto in §4.2.3, memorizzando il blocco, impostando `L1Invalid` a `true` e rispondendo alla L1-X con `WB_ACK` che, una volta ricevuto dalla L1-X, la fa transire nello stato I concludendo di fatto il rimpiazzamento.

4.3.4 Rimpiazzamenti in L2Cache

Come già precedentemente esposto in §4.2.4, se un blocco di cache in L2$ si trova nello stato S e viene selezionato come vittima per un rimpiazzamento dall'algoritmo LRU della L2$ stessa, quest'ultima invalida il blocco senza, di fatto, comunicare nulla alla DirectoryCache (rimpiazzamento silente).

La Figura 15 invece, illustra lo scenario in cui a dover rimpiazzare sia un blocco di cache in L2-X che si trovi nello stato E oppure nello stato M.

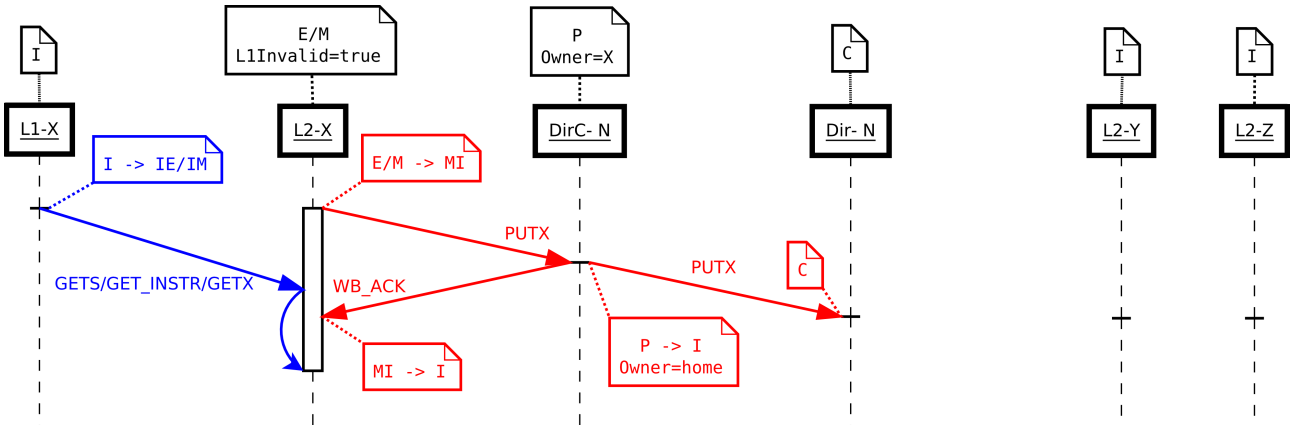

Figura 15 Rimpiazzamento di blocco di memoria non condiviso

Essendo il flag `L1Invalid` impostato a `true`, la L2-X ha la sicurezza che la L1-X è invalidata, quindi può rimpiazzare il blocco di memoria inviandolo, assieme al proprio identificatore, alla Dir$ per mezzo di una `PUTX`, andando poi a transire nello stato MI.

La Dir$ si comporta come descritto in §4.2.4: si trova nello stato P ed ha il campo `Owner` impostato con l'identificatore del banco di L2-X che è l'effettivo “proprietario” del blocco. Non appena riceve il blocco di memoria incapsulato nella `PUTX`, la Dir$ deve, per forza di cose, inoltrarlo alla Directory, utilizzando la *virtual network* ordinata, in quanto non è in grado di memorizzarlo. Invia inoltre, utilizzando sempre la *virtual network* ordinata, il messaggio `WB_ACK` al banco di L2-X che ha rimpiazzato (l'identificatore è contenuto nel messaggio `PUTX`); imposta il campo `Owner` ad `home`, diventando l'effettiva “proprietaria” del blocco; transisce nello stato I.

La Directory, il cui blocco, relativo al blocco di memoria da rimpiazzare, è contenuto nella DirectoryCache, si trova nello stato C. Vedendosi arrivare un blocco di memoria contenuto nel messaggio `PUTX`, non fa altro che memorizzarlo in memoria principale, senza transire di stato.

La L2-X, all'arrivo del `WB_ACK` proveniente dalla Dir$, conclude il rimpiazzamento, facendo transire il blocco di cache nello stato I.

Si noti che a rimpiazzamento in corso, è possibile che arrivino alla L2-X richieste da parte della L1-X. Quindi bisogna prevedere nello stato MI l'arrivo di tali richieste che andranno rimesse in coda per poterle di nuovo schedulare a rimpiazzamento ultimato.

4.4 L2 Miss

Si supponga che un processore X, dovendo effettuare operazioni su un blocco di memoria la cui copia non presente nè in L1-X, nè tantomeno in L2-X, scatena miss in entrambe le cache; per cui bisogna recuperare il blocco in oggetto direttamente dalla memoria principale.

4.4.1 Load oppure iFetch

Si presupponga che il blocco di memoria, la cui copia è assente nei banchi di L2-X, possa essere presente in altri banchi, sempre della L2$, relativi a uno o più processori diversi da X.

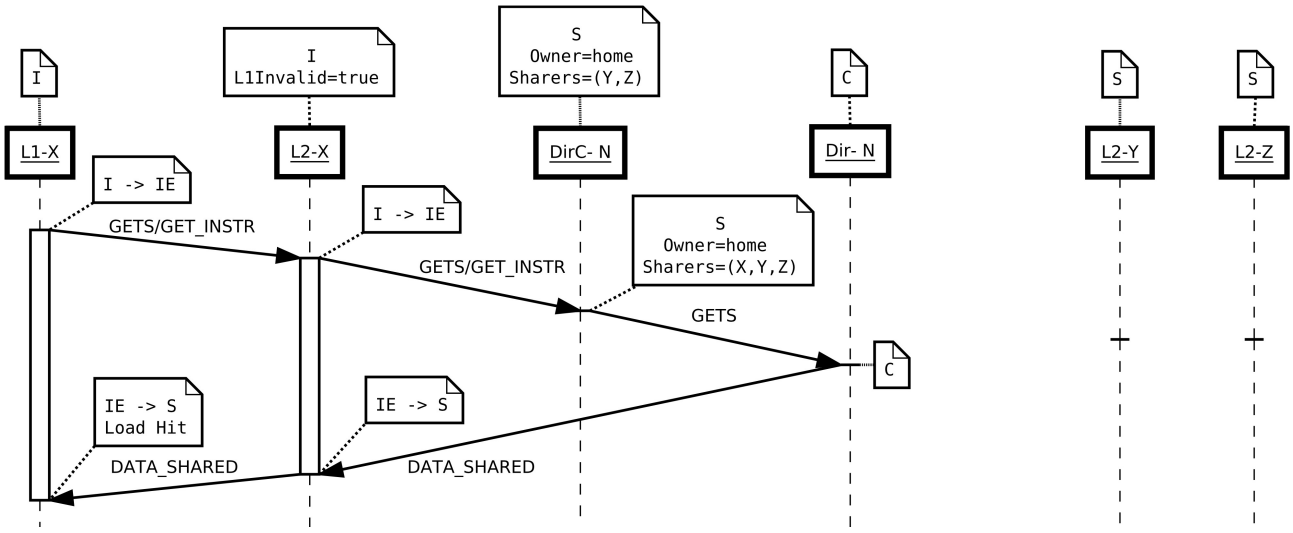

Figura 16 Load oppre iFetch con L2 Miss su blocco condiviso

La Figura 16 illustra lo scenario in cui il blocco di memoria in oggetto è condiviso con i processori Y e Z. Il proprietario del blocco è la Directory che memorizza la lista degli identificatori dei blocchi di L2$ associati ai processori che in precedenza hanno richiesto il blocco.

La L1-X, per effetto della miss, invia il messaggio `GETS/GET_INSTR` verso la L2-X e transisce nello stato IE.

La L2-X si comporta in modo analogo e, non potendo soddisfare la richiesta proveniente dalla L1-X per effetto della miss, invia a sua volta una richiesta del blocco, assieme al proprio identificatore, alla DirectoryCache per mezzo di una GETS/GET_INSTR e transisce nello stato IE.

La Dir$, che fa le veci della Directory, è l'effettiva "proprietaria" del blocco di memoria condiviso che è memorizzato in memoria principale. Per cui necessita di inviare un messaggio di tipo GETS, contenente anche l'identificatore del banco della L2-X richiedente, alla Directory in memoria principale utilizzando la *virtual network* ordinata, ed aggiungere l'identificatore del banco richiedente contenuto nel messaggio alla lista degli Sharers.La Directory vedendosi recapitare una richiesta per il blocco di memoria in modo condiviso, preleva quest'ultimo dalla memoria principale e lo incapsula in un messaggio DATA_SHARED che invierà direttamente al banco della L2-X che ne ha fatto richiesta.

Quando la L2-X riceve il messaggio DATA_SHARED contenente il blocco, lo memorizza, lo reincapsula in un altro messaggio DATA_SHARED da inviare alla L1-X, e transisce nello stato S.

Analogamente la L1-X, ricevendo DATA_SHARED, memorizza il blocco, sblocca il processore permettendo la Load Hit e transisce a sua volta anch'essa nello stato S.

Si supponga adesso che il blocco di memoria non sia condiviso: ossia che nessun processore ne abbia fatto richiesta per una copia. La Directory si trova nello stato I ed è proprietaria del blocco. Tale scenario è illustrato nella Figura 17.

La L1-X invia GETS/GET_INSTR alla L2-X che a sua volta invia GETS/GET_INSTR, contenente anche il proprio identificatore, alla DirectoryCache. Quest'ultima essendo adesso nello stato I, invia una richiesta per il blocco di memoria in modo esclusivo, assieme all'identificatore del richiedente, alla Directory per mezzo di una GETX, inviata per mezzo della *virtual network* ordianata, ed imposta la L2-X richiedente come proprietaria del blocco. Transisce infine nello stato P in quanto ora il blocco di memoria appartiene al processore associato al banco della L2Cache che ne ha fatto richiesta, avendo appositamente impostato il campo Owner col l'apposito identificatore.

La Directory ricevendo il messaggio GETX, preleva il blocco dalla memoria principale e lo incapsula nel meassaggio DATA_EXCLUSIVE, che andrà spedito al banco di L2-X richiedente, senza transire di stato.

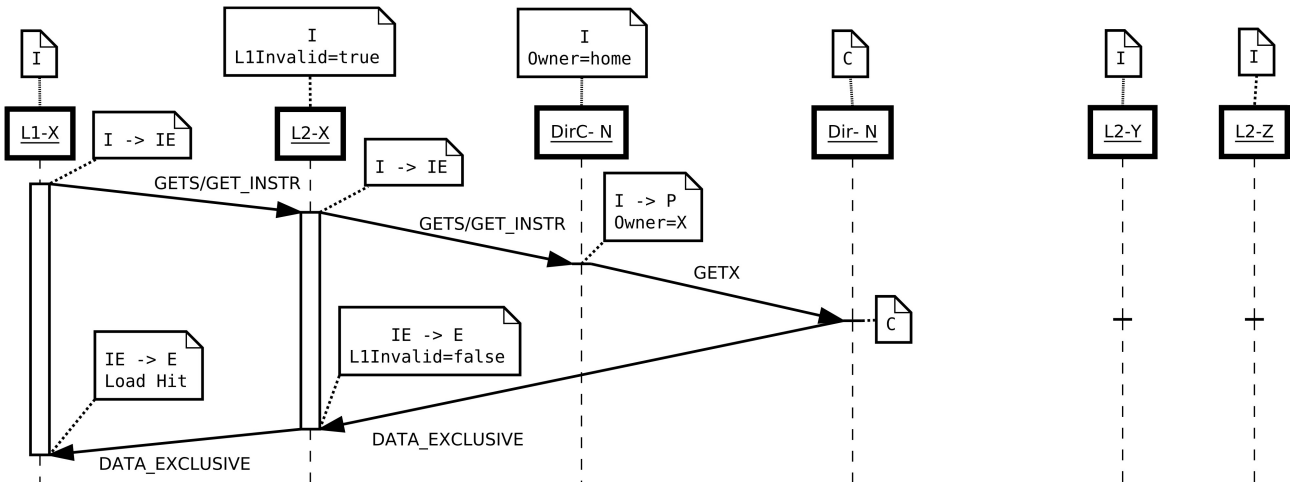

Figura 17 Load oppre iFetch con L2 Miss e blocco non condiviso

La L2-X, ricevendo il blocco di memoria per mezzo del messaggio `DATA_EXCLUSIVE`, lo memorizza, lo inoltra alla L1-X (incapsulato nel messaggio `DATA_EXCLUSIVE`) e transisce nello stato E.

La L1-X, infine, ricevendo il blocco di memoria dalla L2-X, lo memorizza, sblocca il processo permettendo la Load Hit e transisce a sua volta nello stato E.

4.4.2 Store

La Figura 18 illustra lo scenario in cui il processore X debba effettuare un'operazione di Store su un blocco di memoria condiviso dalle cache L2-Y ed L2-Z. La Directory è nello stato S, proprietaria del blocco, con Y e Z nella lista degli sharers.

La L1-X, per effetto della miss, invia un messaggio `GETX` alla L2-X e transisce nello stato IM.

La L2-X non può servire la richiesta pervenutagli dalla L1-X, poichè non possiede il blocco di memoria; quindi inoltra la `GETX`, che adesso conterrà anche il proprio identificatore, alla DirectoryCache e transisce anch'essa nello stato IM.La Dir$, ricevuta la `GETX` dalla L2-X, invia, utilizzando la *virtual Network* ordianata, un messaggio di invalidazione (`INV`) a ciascun banco della L2Cache i cui identificatori sono presenti nella lista degli sharers, svuota tale lista, e imposta la L2-X come proprietaria del blocco. Il messaggio di invalidazione deve contenere anche l'identificatore del banco della L2-X richiedente il blocco. Inoltre, prima di transire nello stato S, la Dir$ invia un

messaggio di richiesta del blocco di memoria condiviso (GETS) alla Directory utilizzando sempre la *virtual network* ordinata. Tale messaggio deve trasportare un'informazione addizionale: il numero dei messaggi di acknowledgement (ACK) che la L2-X richiedente deve attendere prima di considerare le copie del blocco di memoria, che successivamente la Directory le invierà, come invalidate dagli altri banchi della L2$ che precedentemente le condividevano.

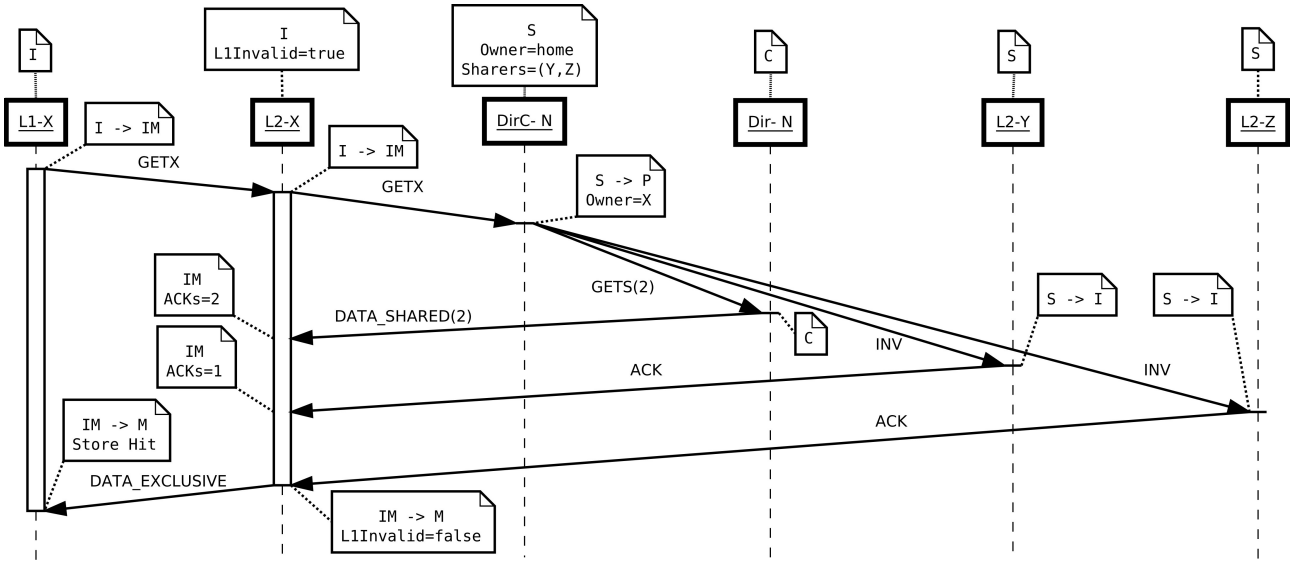

Figura 18 Store con L2 Miss su blocco condiviso

Ogni banco della L2$ che possiede il blocco di memoria condiviso, si vede recapitare un messaggio di invalidazione da parte della Dir$ che contiene l'identificatore del banco della L2-X richiedente il blocco; Per cui ognuna di esse, confeziona un messaggio di acknowledgement (ACK) e lo invia alla L2-X che ha richiesto il blocco per poi transire nello stato I.

La Directory, vedendosi recapitare il messaggio GETS, preleva il blocco dalla memoria principale, lo incapsula assieme al numero degli acknowledgement contenuti nella richiesta ricevuta in un messaggio DATA_SHARED, e lo invia alla L2-X richiedente senza transire di stato.

La L2-X invece, essendo nello stato IM, si vede recapitare il messaggio DATA_SHARED, contenente la copia del blocco di memoria assieme al numero degli acknowledgement da attendere al fine dell'avvenuta invalidazione dei banchi di L2$ che possedevano le copie del blocco condiviso. Quindi memorizza il blocco di memoria ed imposta il campo ACKs (con cui è stato chiamato, per semplicità, il campo NumPendingAcks della TBE) con il numero degli acknowledment contenuti nel messaggio ricevuto. Permane sempre nello stato IM.

Man mano che la L2-X riceve i vari ACK dagli altri banchi della L2$ che possedevano il blocco in oggetto, essa non deve far altro che decrementare il campo ACKs fino a che raggiunga il valore 0. Quando ciò accade, la L2-X può ritenere concluse le varie procedure di invalidazione da parte degli altri banchi della L2$ e quindi confezionare un messaggio DATA_EXCLUSIVE, contenente la copia del blocco di memoria, da inviare alla L1-X e, dopo aver impostato il flag L1Invalid a false, transire nello stato M.

La L1-X infine, ricevuta la copia del blocco di memoria in modo esclusivo dalla L2-X, lo memorizza, sblocca il processore per permettere la Store Hit e transisce anch'essa nello stato M.

Restano da illustrare due scenari rimasti in sospeso: la Store con L1 Hit e la Store con L1 Miss ed L2 Hit entrambe su blocco condiviso. Essi sono analoghi ad una Store con L2 Miss su blocco condiviso appena discussa e vengono illustrati rispettivamente nelle Figure 19 e 20. In esse si può notare anche come i vari messaggi di acknowledgement inviati dai vari banchi della L2$ ed il messaggio contenente la copia del blocco di memoria proveniente dalla Directory possono arrivare alla L2-X in qualsiasi ordine facendo in modo che il contatore ACKs possa assumere anche valori negativi.

La Figura 21 illustra lo scenario in cui il blocco di memoria, neccessario al processore X per effettuare una Store, sia memorizzato in memoria principale (ossia non posseduto da nessun banco di L2$ e quindi appartenente alla Directory) e l'operazione di Store provochi miss sia in L1-X sia in L2-X.

La L1-X, per effetto della miss, invia GETX alla L2-X e transisce nello stato IM. La L2-X, analogamente, invia GETX contenente il proprio identificatore alla Dir$ e transisce anch'essa nello stato IM.

La DirectoryCache, proprietaria del blocco di memoria, imposta l'Owner ad X, confeziona il messaggio GETX (contenente anche l'identificatore del banco della L2-X richiedente), lo invia alla Directory tramite la *virtual network* ordinata e tansisce nello stato P.La Directory, come noto, ricevendo una GETX dalla Dir$, preleva il blocco dalla memoria principale, lo incapsula in un messaggio DATA_EXCLUSIVE e lo invia alla L2-X richiedente restando nello stato C.

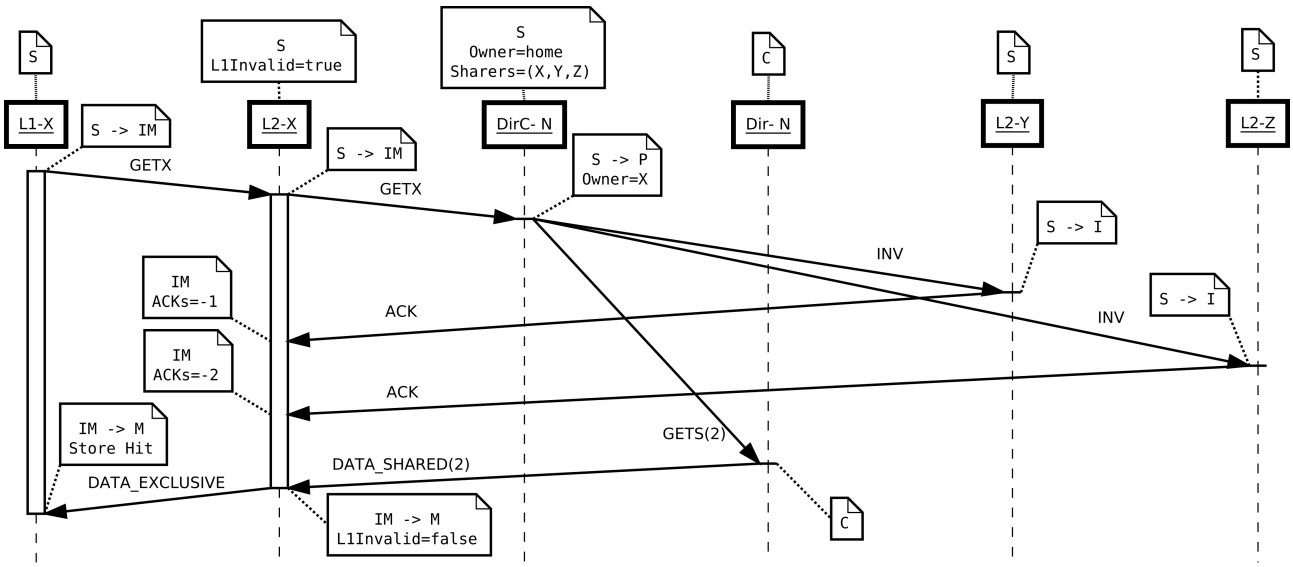

Figura 19 Store con L1 Hit su blocco condiviso

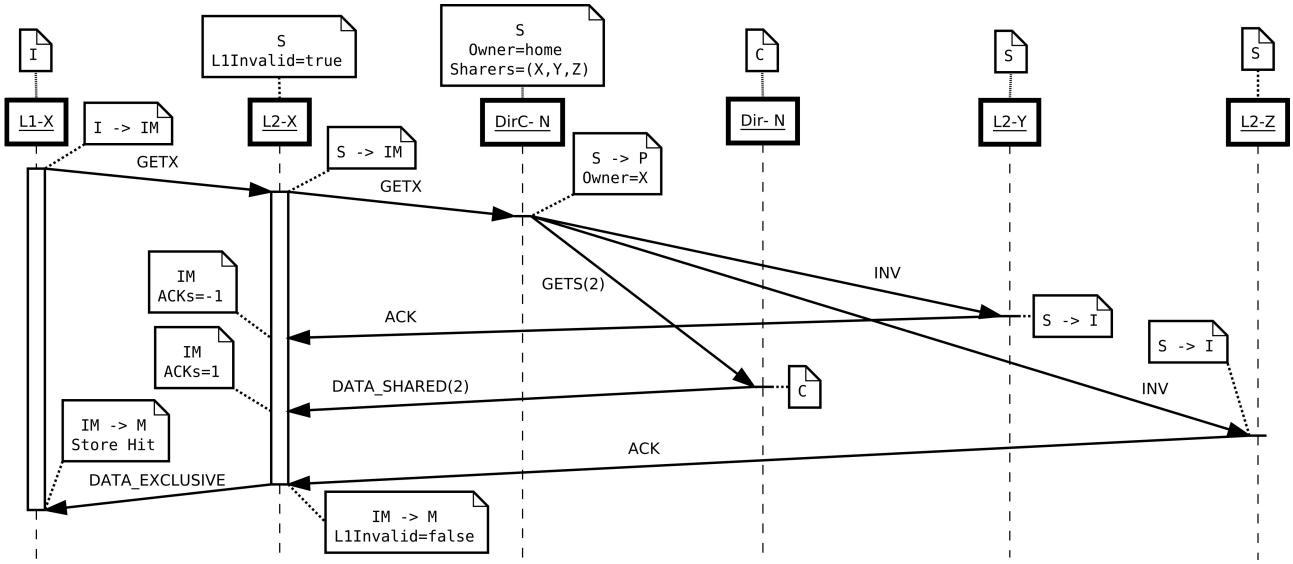

Figura 20 Store con L1 Miss ed L2 Hit su blocco condiviso

La L2-X, essendo nello stato IM e ricevendo il `DATA_EXCLUSIVE` proveniente dalla Directory, memorizza il blocco ricevuto, imposta il flag `L1Invalid` a `false`, confeziona ed invia un nuovo messaggio `DATA_EXCLUSIVE` per la L1-X e transisce nello stato M.

La L1-X infine, ricevuta la copia del blocco di memoria in modo esclusivo dalla L2-X, lo memorizza, sblocca il processore per permettere la Store Hit e transisce anch'essa nello stato M.

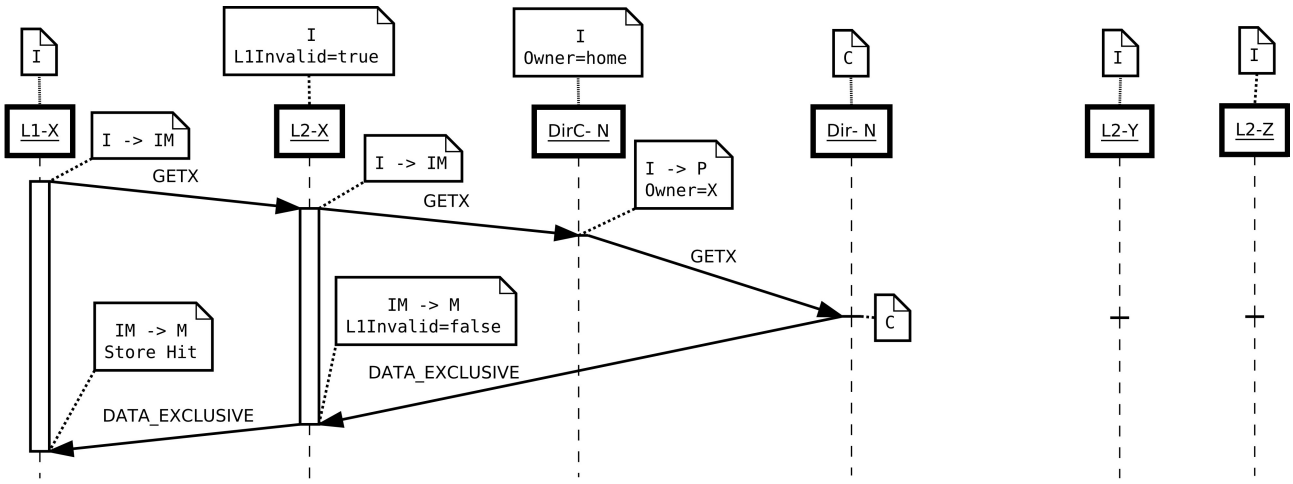

Figura 21 Store con L2 Miss su blocco non condiviso

4.4.3 Rimpiazzamenti in L2Cache

La Figura 22 illustra lo scenario in cui un rimpiazzamento per un blocco di cache in L2-X venga scatenato quando tale blocco è nello stato IE: ossia dopo che la L2-X stessa abbia inviato la `GETS/GET_INSTR` alla DirectoryCache, per un blocco di memoria condiviso, e prima che abbia ricevuto il `DATA_SHARED` da quest'ultima.

La L1-X, per effetto della miss, invia il messaggio `GETS/GET_INSTR` verso la L2-X e transisce nello stato IE. La L2-X, non potendo soddisfare la richiesta proveniente dalla L1-X, invia a sua volta una richiesta del blocco, assieme al proprio identificatore, alla DirectoryCache per mezzo di `GETS/GET_INSTR` e transisce nello stato IE. La Dir$, che fa le veci della Directory, è l'effettiva "proprietaria" del blocco di memoria condiviso che è memorizzato in memoria principale. Per cui necessita di inviare un messaggio di tipo `GETS`, contenente anche l'identificatore del banco della L2-X richiedente, alla Directory in memoria principale utilizzando la *virtual network* ordinata, ed aggiungere l'identificatore contenuto nel messaggio alla lista degli `Sharers`. La Directory vedendosi recapitare una richiesta per il blocco di memoria in modo condiviso, preleva quest'ultimo dalla memoria principale e lo incapsula in un messaggio `DATA_SHARED` che invierà direttamente al banco della L2-X che ne ha fatto richiesta.

Prima che arrivi il `DATA_SHARED`, contenente la copia del blocco di memoria richiesto, il blocco di cache in L2-X viene scelto dall'algoritmo LRU come vittima per un rimpiazzamento. Per cui la L2-X reagisce inviando un messaggio `GETX` alla L1-X, in modo da provocare in quest'ultima

un rimpiazzamento “forzato”, ed andando a transire nello stato IEI.

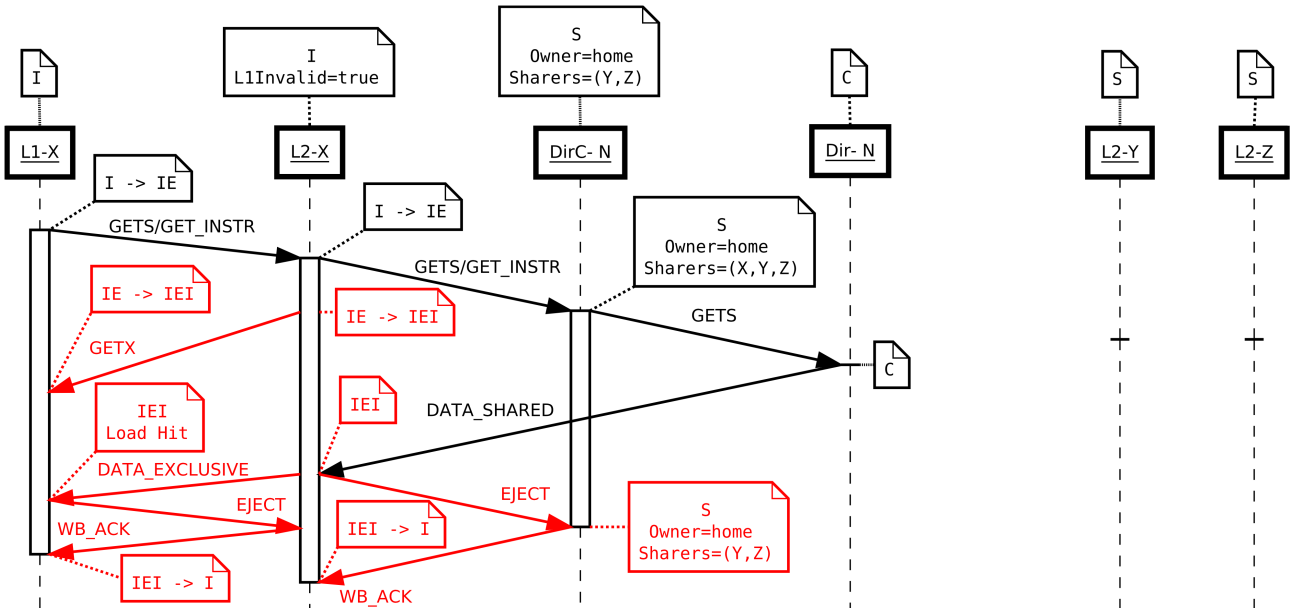

Figura 22 Rimpiazzamento post Load/iFetch per blocco di memoria condiviso

La L1-X, a seguito della ricezione della `GETX` dalla L2-X, transisce anch'essa nello stato IEI.

La L2-X, alla ricezione del `DATA_SHARED` proveniente dalla Directory e contenente la copia del blocco di memoria, lo memorizza, lo reincapsula in un messaggio `DATA_EXCLUSIVE` da inviare alla L1-X ed inizia la fase di rimpiazzamento inviando un messaggio `EJECT` verso la Dir$ senza di fatto transire di stato.

La Dir$, ricevuto il messaggio `EJECT`, senza transire di stato elimina l'identificatore del mittente dalla lista degli sharers e risponde alla L2-X inviandole il `WB_ACK` utilizzando la *virtual network* ordinata che, una volta ricevuto, la fa transire nello stato I, concludendo di fatto il rimpiazzamento.

La L1-X, alla ricezione del `DATA_EXCLUSIVE` contenente la copia del blocco di memoria, lo memorizza, sblocca il processore per permettere la Load Hit ed invia il messaggio `EJECT` alla L2$, iniziando, di fatto, la fase di rimpiazzamento, senza transire di stato.

La L2-X, ricevuto il messaggio `EJECT`, senza transire di stato risponde alla L1-X inviandole il `WB_ACK` che, una volta ricevuto dalla L1-X stessa, la fa transire nello stato I concludendo di fatto il rimpiazzamento. Si noti che il messaggio EJECT può arrivare anche quando la L2-X, a seguito dell'arrivo del WB_ACK proveniente dalla Dir$, sia già transitata nello stato I.

La Figura 23 illustra lo scenario precedente, in cui un rimpiazzamento per un blocco di cache in L2-X venga scatenato quando tale blocco è nello stato IE: ossia dopo che la L2-X stessa abbia inviato la `GETS/GET_INSTR` alla DirectoryCache, per un blocco di memoria adesso non condiviso, e prima che abbia ricevuto il `DATA_EXCLUSIVE` da quest'ultima.

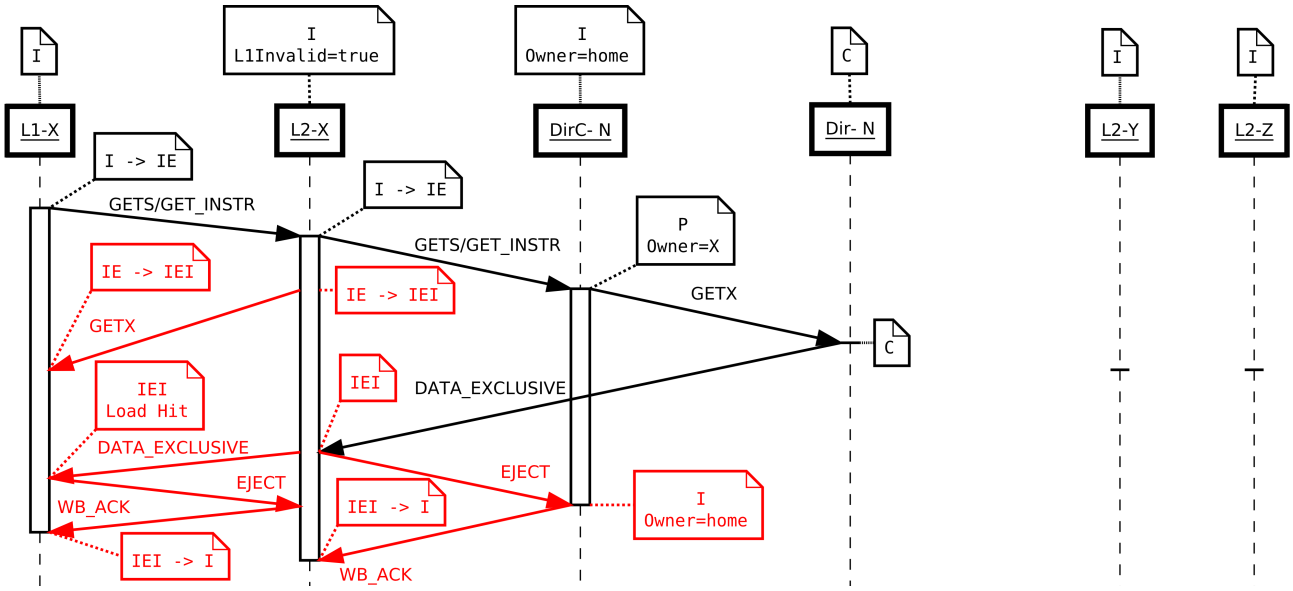

Figura 23 Rimpiazzamento post Load/iFetch per blocco di memoria non condiviso

La L1-X, per effetto della miss, invia il messaggio `GETS/GET_INSTR` verso la L2-X e transisce nello stato IE. La L2-X, non potendo soddisfare la richiesta proveniente dalla L1-X, invia a sua volta una richiesta del blocco di memoria, assieme al proprio identificatore, alla DirectoryCache per mezzo di `GETS/GET_INSTR` e transisce nello stato IE. La Dir$, che fa le veci della Directory, è l'effettiva "proprietaria" del blocco di memoria non condiviso che è memorizzato nella memoria principale. Per cui necessita di inviare un messaggio di tipo `GETX`, contenente anche l'identificatore del banco della L2-X richiedente, alla Directory in memoria principale utilizzando la *virtual network* ordinata, ed impostare il campo `Owner` con l'identificatore contenuto nel messaggio, facendo diventare la L2-X la nuova "proprietaria" del blocco. La Directory vedendosi recapitare una richiesta per il blocco di memoria in modo non condiviso, preleva quest'ultimo dalla memoria principale e lo incapsula in un messaggio `DATA_EXCLUSIVE` che invierà direttamente al banco della L2-X che ne ha fatto richiesta.

Prima che arrivi il `DATA_EXCLUSIVE`, contenente la copia del blocco di memoria richiesto, il blocco di cache in L2-X viene scelto dall'algoritmo LRU come vittima per un rimpiazzamento. Per cui la L2-X reagisce inviando un messaggio `GETX` alla L1-X, in modo da provocare in quest'ultima

un rimpiazzamento “forzato”, ed andando a transire nello stato IEI.

La L1-X, a seguito della ricezione della GETX dalla L2-X, transisce anch'essa nello stato IEI.

La L2-X, alla ricezione del DATA_EXCLUSIVE proveniente dalla Directory e contenente la copia del blocco di memoria, lo memorizza, lo reincapsula in un altro messaggio DATA_EXCLUSIVE da inviare alla L1-X ed inizia la fase di rimpiazzamento inviando un messaggio EJECT verso la Dir$ senza di fatto transire di stato.

La Dir$, ricevuto il messaggio EJECT, imposta il campo Owner con home, diventando l'effettiva “proprietaria” del blocco, invia alla L2-X il WB_ACK, utilizzando la *virtual network* ordinata, e transisce nello stato I.

La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce nello stato I concludendo di fatto il rimpiazzamento.

La L1-X, alla ricezione del DATA_EXCLUSIVE contenente la copia del blocco di memoria, lo memorizza, sblocca il processore per permettere la Load Hit ed invia il messaggio EJECT alla L2-X, iniziando, di fatto, la fase di rimpiazzamento, senza transire di stato.

La L2-X, ricevuto il messaggio EJECT, senza transire di stato risponde alla L1-X inviandole il WB_ACK che, una volta ricevuto dalla L1-X stessa, la fa transire nello stato I concludendo di fatto il rimpiazzamento.

La Figura 24 illustra lo scenario in cui un rimpiazzamento per un blocco di cache in L2-X venga scatenato quando tale blocco è nello stato IM: ossia dopo che la L2-X stessa abbia inviato la GETX alla DirectoryCache, per un blocco di memoria condiviso, e prima che abbia ricevuto il DATA_SHARED, contenente il blocco di memoria, dalla Directory e comunque prima che arrivino i susseguenti ACK provenienti dagli altri banchi di L2$ che possedevano una copia del blocco.

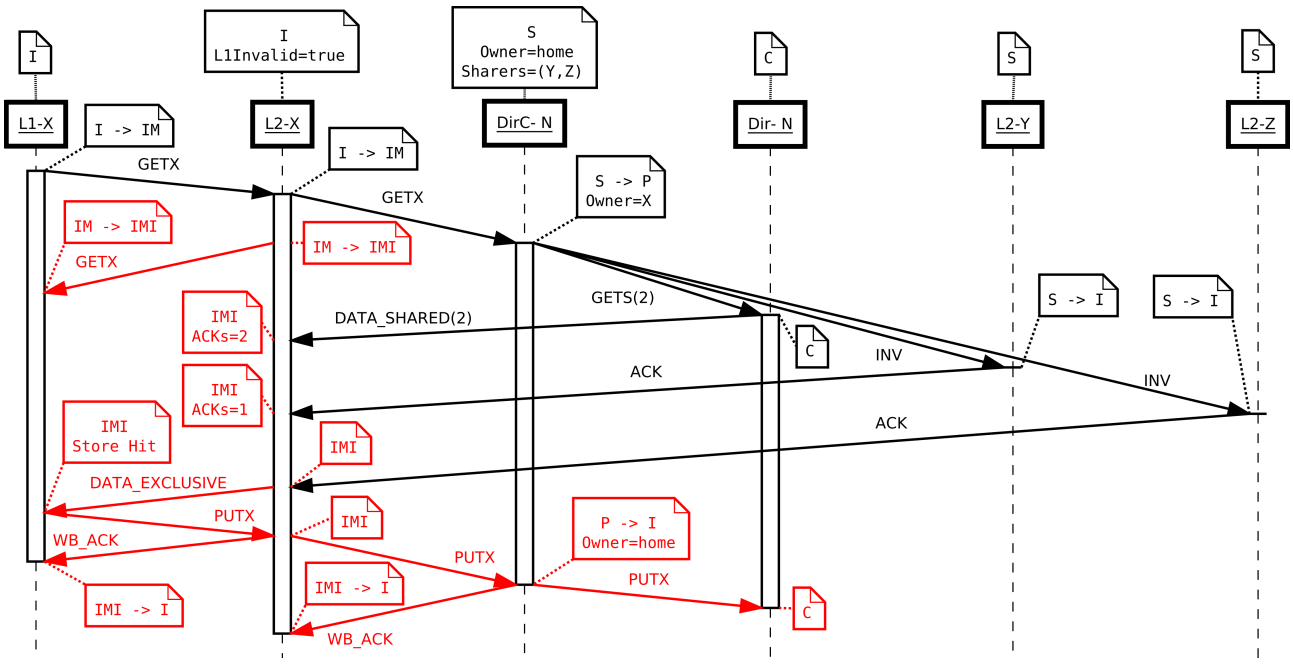

Figura 24 Rimpiazzamento post Store per blocco di memoria condiviso

Come illustrato in §4.4.2, la L1-X, per effetto della miss, invia un messaggio `GETX` alla L2-X e transisce nello stato IM. La L2-X non può servire la richiesta pervenutagli dalla L1-X, poichè possiede il blocco di memoria in modo condiviso; quindi inoltra la richiesta `GETX`, contenente il proprio identificatore, alla DirectoryCache e transisce anch'essa nello stato IM.

La Dir$, ricevuta la `GETX` dalla L2-X, invia, utilizzando la *virtual network* ordinata, un messaggio di invalidazione (`INV`) a ciascun banco della L2Cache i cui identificatori sono presenti nella lista degli sharers, svuota tale lista, e imposta la L2-X come proprietaria del blocco.

Il messaggio di invalidazione deve contenere anche l'identificatore del banco della L2-X richiedente il blocco. Inoltre, prima di transire nello stato S, la Dir$ invia un messaggio di richiesta del blocco di memoria condiviso (`GETS`) alla Directory utilizzando sempre la *virtual network* ordinata. Tale messaggio deve trasportare un'informazione addizionale: il numero dei messaggi di acknowledgement (`ACK`) che la L2-X richiedente deve attendere prima di considerare le copie del blocco di memoria, che successivamente la Directory le invierà, come invalidate dagli altri banchi della L2$ che precedentemente le condividevano. Ogni banco della L2$ che possiede il blocco di memoria condiviso, si vede recapitare un messaggio di invalidazione da parte della Dir$ che contiene l'identificatore del banco della L2-X richiedente il blocco; Per cui ognuna di esse, confeziona un messaggio di acknowledgement (`ACK`) e lo invia alla L2-X che ha richiesto il blocco

per poi transire nello stato I. La Directory, vedendosi recapitare il messaggio GETS, preleva il blocco dalla memoria principale, lo incapsula in in messaggio DATA_SHARED, contenente anche il numero degli acknowledgement contenuti nella richiesta ricevuta, e lo invia alla L2-X richiedente senza transire di stato.

Si supponga che, prima che la L2-X abbia ricevuto il DATA_SHARED assieme a tutti gli ACK provenienti dagli altri banchi di L2$ che possedevano una copia del blocco, il blocco di cache venga selezionato come vittima per un rimpiazzamento dall'algoritmo LRU della L2$. Per cui la L2-X reagisce inviando un messaggio GETX alla L1-X ed andando a transire nello stato IMI.

La L1-X, a seguito della ricezione della GETX dalla L2-X, transisce anch'essa nello stato IMI.

La L2-X invece, permanendo nello stato IMI, si vede recapitare il messaggio DATA_SHARED, contenente la copia del blocco di memoria assieme al numero degli acknowledgement da attendere al fine dell'avvenuta invalidazione dei banchi di L2$ che possevano le copie del blocco condiviso. Quindi memorizza il blocco di memoria ed imposta il campo ACKs con il numero degli acknowledment contenuti nel messaggio ricevuto.

Man mano che la L2-X riceve i vari ACK dagli altri banchi della L2$ che possedevano il blocco in oggetto, essa non deve far altro che decrementare il campo ACKs fino a che esso raggiunga il valore 0. Quando ciò accade, la L2-X può ritenere concluse le varie procedure di invalidazione da parte degli altri banchi della L2$ e quindi confezionare un messaggio DATA_EXCLUSIVE, contenente la copia del blocco di memoria, da inviare alla L1-X, permamendo ancora in IMI.

La L1-X, ricevuta la copia del blocco di memoria in modo esclusivo dalla L2-X, lo memorizza, sblocca il processore per permettere la Store Hit ed inizia la propria fase di rimpiazzamento inviando la copia del blocco di memoria, modificato dalla Store, alla L2-X per mezzo del messaggio PUTX; permanendo in IMI.

La L2-X, ricevuto il blocco di memoria modificato dalla L1-X, lo memorizza; lo inoltra per mezzo di un messaggio PUTX, contenente il proprio identificatore, alla DirectoryCache, iniziando anch'essa la propria fase di rimpiazzamento; ed infine invia il WB_ACK alla L1-X che, una volta ricevutolo, conclude il rimpiazzamento andando a transire nello stato I.

La Dir$, ricevuto il messaggio PUTX da parte della L2-X che è l'attuale “proprietaria” del blocco di memoria, imposta il campo Owner con home, diventando di fatto l'effettiva “proprietaria” del blocco; invia alla L2-X il WB_ACK utilizzando la *virtual network* ordinata e, non avendo la possibilità di memorizzare il blocco di memoria, lo invia, mediante una PUTX sulla *virtual network* ordinata, alla Directory, andando infine a transire nello stato I.

La Directory, ricevuto il messaggio PUTX, memorizza il blocco in memoria principale senza transire di stato.

La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce nello stato I concludendo di fatto il rimpiazzamento.

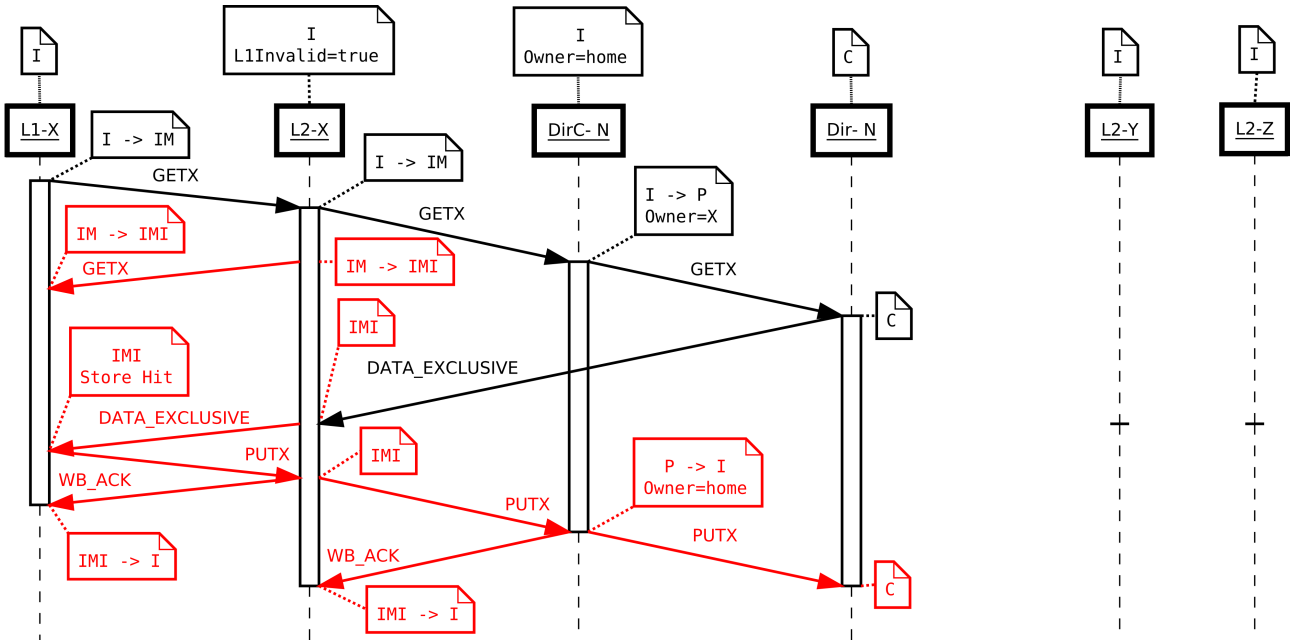

Figura 25 Rimpiazzamento post Store per blocco di memoria non condiviso

La Figura 25 illustra lo scenario in cui un rimpiazzamento per un blocco di cache in L2-X venga scatenato quando tale blocco è nello stato IM: ossia dopo che la L2-X stessa abbia inviato la GETX alla DirectoryCache, per un blocco di memoria non condiviso, e prima che abbia ricevuto il DATA_EXCLUSIVE, contenente il blocco di memoria, da parte della Directory.Come illustrato in §4.4.2, la L1-X, per effetto della miss, invia un messaggio GETX alla L2-X e transisce nello stato IM. Analogamente la L2-X, sempre per effetto della miss, invia un messaggio GETX, contenente il proprio identificatore, alla DirectoryCache e transisce anch'essa nello stato IM. La Dir$, ricevuta la

GETX dalla L2-X, imposta la L2-X come proprietaria del blocco, invia un messaggio di richiesta del blocco di memoria non condiviso (GETX) alla Directory utilizzando la *virtual network* ordinata e transisce nello stato P. La Directory, vedendosi recapitare il messaggio GETX, preleva il blocco dalla memoria principale, lo incapsula in in messaggio DATA_EXCLUSIVE, e lo invia alla L2-X richiedente senza transire di stato.

Si supponga che, prima che la L2-X abbia ricevuto il DATA_EXCLUSIVE, il blocco di cache venga selezionato come vittima per un rimpiazzamento dall'algoritmo LRU della L2$. Per cui la L2-X reagisce inviando un messaggio GETX alla L1-X, in modo da provocare in quest'ultima un rimpiazzamento “forzato”, ed andando a transire nello stato IMI.

La L1-X, a seguito della ricezione della GETX dalla L2-X, transisce anch'essa nello stato IMI.

La L2-X invece, permanendo nello stato IMI, si vede recapitare il messaggio DATA_EXCLUSIVE, contenente la copia del blocco di memoria non condiviso, lo memorizza e lo invia alla L1-X sempre incapsulato nel messaggio DATA_EXCLUSIVE.

La L1-X, ricevuta la copia del blocco di memoria in modo esclusivo dalla L2-X, lo memorizza, sblocca il processore per permettere la Store Hit ed inizia la propria fase di rimpiazzamento inviando la copia del blocco di memoria, modificato dalla Store, alla L2-X per mezzo del messaggio PUTX; permanendo in IMI.

La L2-X, ricevuto il blocco di memoria modificato dalla L1-X, lo memorizza; lo inoltra per mezzo di un messaggio PUTX, contenente il proprio identificatore, alla DirectoryCache, iniziando anch'essa la propria fase di rimpiazzamento; ed infine invia il WB_ACK alla L1-X che, una volta ricevutolo, conclude il rimpiazzamento andando a transire nello stato I.

La Dir$, ricevuto il messaggio PUTX da parte della L2-X che è l'attuale “proprietaria” del blocco di memoria, imposta il campo Owner con home, diventando di fatto l'effettiva “proprietaria” del blocco; invia alla L2-X il WB_ACK utilizzando la *virtual network* ordinata e, non avendo la possibilità di memorizzare il blocco di memoria, lo invia attraverso la PUTX alla Directory, utitizzando sempre la *virtual network* ordinata, andando infine a transire nello stato I.

La Directory, ricevuto il messaggio PUTX, memorizza il blocco in memoria principale senza transire di stato.

La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce nello stato I concludendo di fatto il rimpiazzamento.

4.5 L2 Miss concorrenti

Verrà discusso adesso come si comporta il protocollo quando processori necessitano di effettuare operazioni in modo concorrente su un unico blocco di memoria non condiviso ed appartenente alla Directory.

4.5.1 Load oppure iFetch multiple

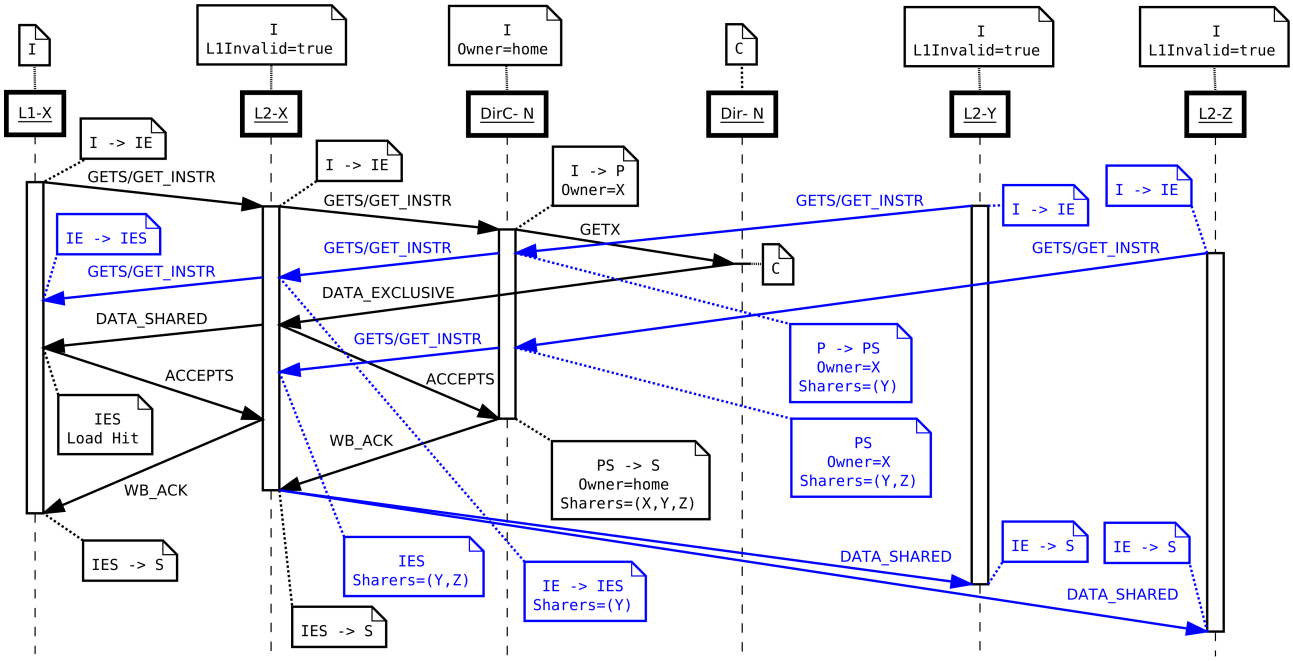

Figura 26 Load oppure iFetch multiple

La Figura 26 illustra lo scenario in cui più processori necessitano di effettuare operazioni di Load oppure iFetch in modo concorrente su un blocco di memoria non condiviso ed appartenente alla Directory.

La L1-X invia GETS/GET_INSTR alla L2-X e transisce nello stato IE. La L2-X analogamente invia GETS/GET_INSTR, contente il proprio identificatore, alla DirectoryCache e transisce anch'essa nello stato IE. La Dir$, essendo proprietaria del blocco di memoria non condiviso, all'arrivo della richiesta salla L2-X, confeziona ed invia una richiesta GETX, contenente anche l'identificatore del banco di L2$ richiedente, verso la Directory utilizzando la *virtual network* ordinata; imposta l'Owner con l'identificatore del banco della L2$ richiedente; transisce nello stato P. La Directory, ricevendo una richiesta per un blocco di memoria in modo esclusivo dalla Dir$, preleva il blocco dalla memoria principale, lo incapsula nel messaggio DATA_EXCLUSIVE, e lo invia alla L2-X richiedente senza transire di stato.

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco in oggetto provenienti da banchi di L2$ associati ad altri processori che non siano X. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata alla L2-X, la Dir$ aggiunge l'identificatore dalla nuova L2Cache richiedente alla lista degli sharers e transisce nello stato PS.

Si supponga inoltre che la L2-X riceva la GETS/GET_INSTR inoltratagli dalla Dir$ prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve aggiungere l'identificatore del richiedente in una propria lista degli sharers (per semplicità è stata chiamata anch'essa Sharers, ma corrisponde al campo ForwardGetS_IDs della TBE nella L2Cache); inoltre essa invia GETS/GET_INSTR verso la L1-X per poi transire in un nuovo stato: IES (Invalid to Exclusive to Shared) che tiene conto del fatto che altri processori richiedono lo stesso blocco di memoria in modo non esclusivo. Per ogni ulteriore richiesta GETS/GET_INSTR proveniente dalla Dir$, la L2-X deve solo memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato. La L1-X vedendosi recapitare una GETS/GET_INSTR, non fa altro che transire anch'essa in IES.

Quando arriva il DATA_EXCLUSIVE proveniente dalla Directory, la L2-X inoltra il blocco di memoria in esso contenuto alla L1-X per mezzo di un messaggio DATA_SHARED e confeziona un nuovo messaggio di tipo ACCEPTS (Accept Shared), contenente anche il proprio identificatore, da inviare verso la Dir$. La L1-X, in modo analogo, si vede recapitare un DATA_SHARED e, dopo aver memorizzato la copia del blocco di memoria in esso contenuto e, sbloccato il processore per

permettere la Load Hit, invia anch'essa un messaggio di tipo ACCEPTS verso la L2-X.

La Dir$, appena riceve il messaggio ACCEPTS dalla L2-X (proprietaria del blocco di memoria), aggiunge l'Owner alla lista degli sharers; imposta l'Owner ad home, diventando di fatto lei la proprietaria del blocco; confeziona ed invia un messaggio di tipo WB_ACK (Write-Back Acknowledgement) verso la L2-X utilizzando la *virtual network* ordinata, per poi transire nello stato S.

Analogamente la L2-X, non appena arriva l'ACCEPTS proveniente dalla L1-X, le risponde con un WB_ACK, senza però transire di stato.

Non appena arriva il WB_ACK dalla Dir$, la L2-X invia la copia del blocco di memoria, per mezzo di messaggi DATA_SHARED, ad ogni banco di L2Cache il cui identificatore è contenuto nella lista degli sharers; svuota tale lista e transisce nello stato S. La L1-X, vedendosi recapitare il WB_ACK dalla L2-X, transisce anch'essa nello stato S e tutti gli altri banchi di L2Cache richiedenti il blocco passeranno nello stato S non appena riceveranno la copia del blocco di memroia richiesto attraverso i messaggi DATA_SHARED inviategli dalla L2-X come già illustrato in §4.4.1.

Si noti:

- che il messaggio ACCEPTS proveniente dalla L1-X potrebbe arrivare alla L2-X dopo l'arrivo del WB_ACK proveniente dalla DirectoryCache, in tal caso bisogna prevedere questo evento anche quando una L2Cache si trova nello stato S.

- che l'arrivo del WB_ACK impedisce che alla L2-X arrivino altre richieste GETS/GET_INSTR poichè viaggiano tutte su una *virtual network* ordinata e sono tutte richieste precedenti al WB_ACK che rende la Directory proprietaria del blocco.

4.5.2 Load oppure iFetch e Store

La Figura 27 illustra lo scenario in cui un processore X necessita di effettuare un operazione di Load oppure iFetch su un blocco di memoria non condiviso ed appartenente alla Directory, mentre un processore Y necessita di effettuare un'operazione di Store in modo concorrente sullo stesso

blocco.

La L1-X invia GETS/GET_INSTR alla L2-X e transisce nello stato IE. La L2-X analogamente invia GETS/GET_INSTR, contenente il proprio identificatore, alla DirectoryCache e transisce anch'essa nello stato IE. La Dir$, essendo proprietaria del blocco di memoria non condiviso, confeziona ed invia una richiesta GETX, contenente l'identificatore del richiedente, verso la Directory; inoltre imposta l'Owner con l'identificatore della L2-X e transisce nello stato P. La Directory, ricevendo una richiesta di blocco di memoria esclusivo proveniente dalla Dir$, preleva il blocco dalla memoria principale, lo incapsula nel messaggio DATA_EXCLUSIVE, e lo invia alla L2-X richiedente senza transire di stato.

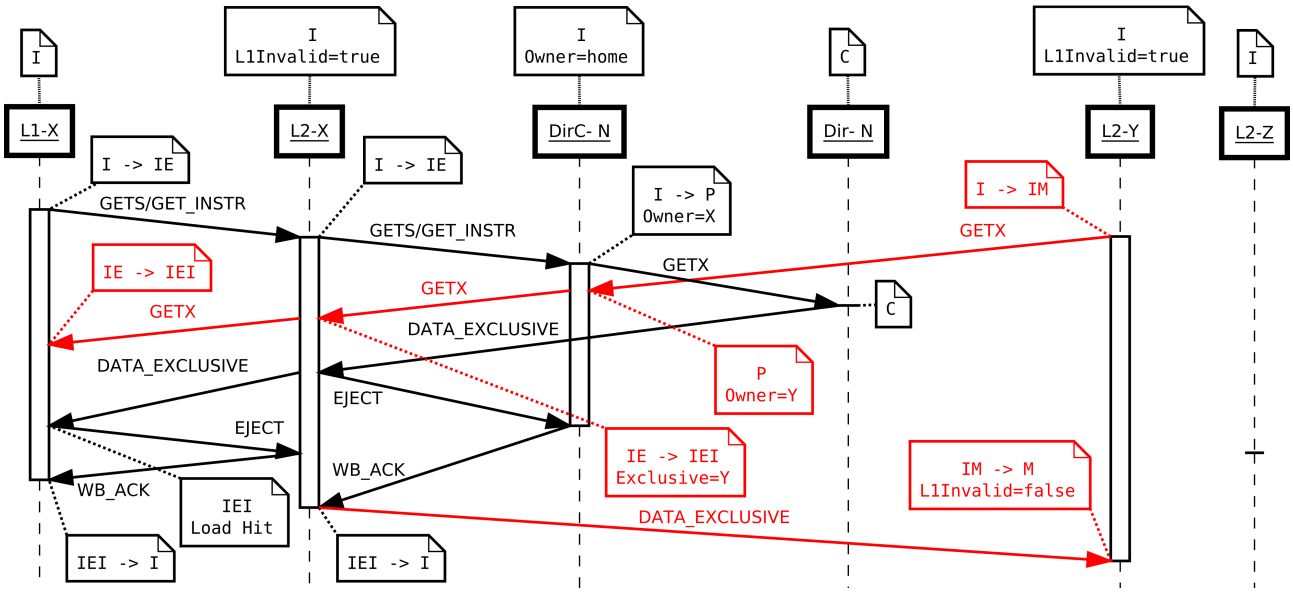

Figura 4-27 Load oppure iFetch e Store

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivi un'ulterio richiesta GETX per il blocco in oggetto proveniente dalla L2-Y. La Dir$ inoltra tale richiesta verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata ed imposta l'Owner con l'identificatore del banco della L2$ richiedente, non modificando lo stato.

Si supponga anche che la L2-X riceva la GETX inoltratagli dalla Dir$ prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve memorizzare l'identificatore del richiedente nel campo Exclusive (che sintetizza, per semplicità, i due campi ForwardGetX_ID e ForwardGetX_ID_present presenti nella TBE della L2$); inoltre

essa invia GETX verso la L1-X per poi transire in IEI: poichè un altro banco della L2$ richiede lo stesso blocco in modo esclusivo.

La L1-X, vedendosi recapitare una GETX, non fa altro che transire anch'essa in IEI.

Quando arriva il DATA_EXCLUSIVE proveniente dalla Directory, la L2-X inoltra la copia del blocco di memoria in esso contenuto alla L1-X sempre per mezzo di un messaggio DATA_EXCLUSIVE e confeziona un nuovo messaggio di tipo EJECT, contenente anche il proprio identificatore, da inviare verso la Dir$. La L1-X, in modo analogo, si vede recapitare un DATA_EXCLUSIVE e, dopo aver memorizzato la copia del blocco di meoria in esso contenuto e, sbloccato il processore per permettere la Load Hit, invia anch'essa un messaggio di tipo EJECT verso la L2-X.

Non appena arriva il messaggio EJECT dalla L2-X, la Dir$ confeziona ed invia un messaggio WB_ACK verso la L2-X utilizzando la *virtual network* ordinata.

Non appena arriva il messaggio EJECT proveniente dalla L1-X, la L2-X le risponde inviandole il WB_ACK, mentre non appena arriva il WB_ACK dalla Dir$, la L2-X invia la copia del blocco di memoria, incapusulato nel messaggio DATA_EXCLUSIVE, alla L2-Y, il cui identificatore è contenuto nel campo Exclusive, finendo per transire nello stato I. Analogamente la L1-X, vedendosi recapitare il WB_ACK dalla L2-X, transisce anch'essa nello stato I mentre la L2-Y richiedente il blocco escusivo passerà nello stato M non appena riceverà il messaggio DATA_EXCLUSIVE inviatogli dalla L2-X come descritto in §4.4.2.

Si noti che il messaggio EJECT proveniente dalla L1-X potrebbe arrivare alla L2-X dopo l'arrivo del WB_ACK proveniente dalla DirectoryCache, in tal caso bisogna prevedere questo evento anche quando un blocco di L2Cache si trova nello stato I.

4.5.3 Load oppure iFetch multiple e Store

La Figura 28 illustra lo scenario in cui più processori necessitano di effettuare operazioni di Load oppure iFetch in modo concorrente su un blocco di memoria non condiviso, appartenente alla Directory, ed un altro processore necessita di effettuare un'operazione di Store sullo stesso blocco.

La L1-X invia GETS/GET_INSTR alla L2-X e transisce nello stato IE. La L2-X analogamente invia GETS/GET_INSTR alla Dir$ insieme al proprio identificatore e transisce anch'essa nello stato IE. La Dir$, essendo proprietaria del blocco di memoria non condiviso, confeziona ed invia una richiesta GETX, contenente l'identificatore del richiedente, verso la Directory; inoltre imposta l'Owner con l'identificatore del banco della L2-X richiedente e transisce nello stato P. La Directory, ricevendo la GETX dalla Dir$, preleva il blocco dalla memoria principale, lo incapsula nel messaggio DATA_EXCLUSIVE, e lo invia alla L2-X richiedente senza transire di stato.

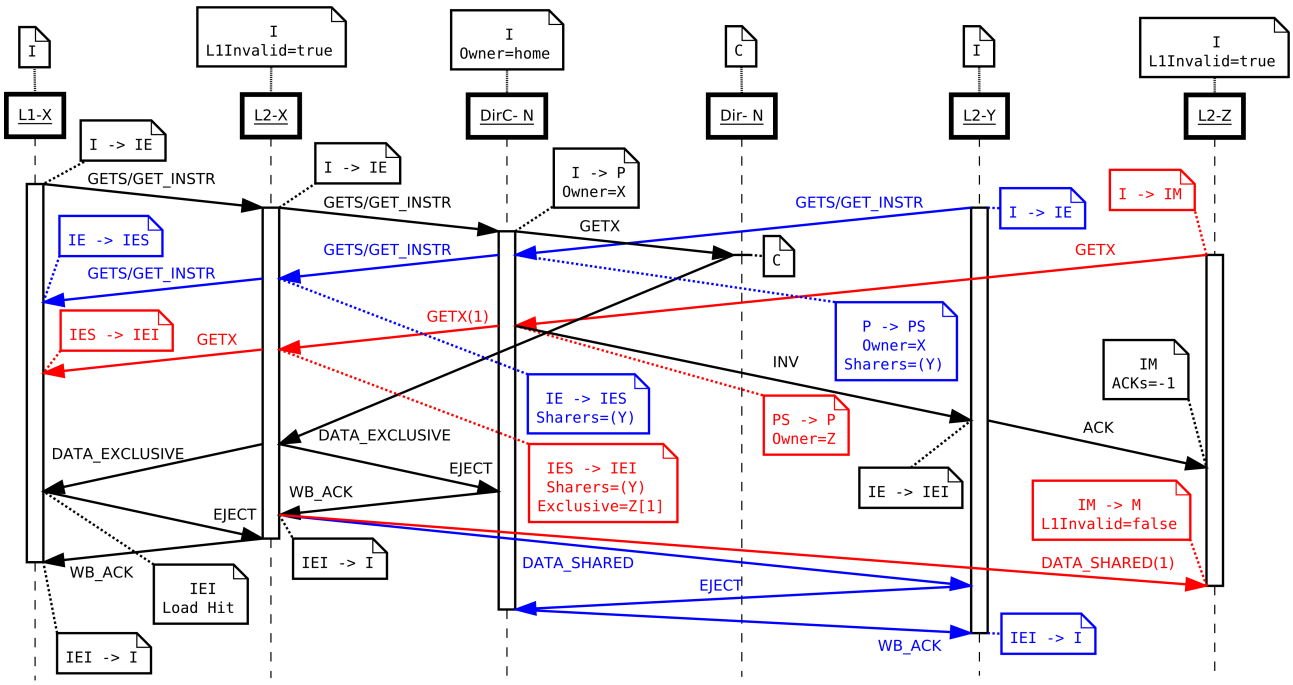

Figura 28 Load oppure iFetch multiple e Store

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco in oggetto. Come già descritto in §4.5.1, la Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata alla L2-X proprietaria del blocco, la Dir$ aggiunge l'identificatore del banco della L2Cache richiedente alla lista degli sharers e transisce nello stato PS.

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivi una richiesta GETX per il blocco in oggetto. La Dir$ inoltra tale richiesta, che contiene anche il numero degli identificatori dei banchi di L2$ presenti nella lista degli sharers,

verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata; imposta l'`Owner` con l'identificatore della nuova L2Cache richiedente (Z); invia ad ogni banco di L2$, il cui identificatore è presente nella lista degli sharers, un messaggio di tipo `INV`, contenente al suo interno anche l'identificatore del nuovo proprietario del blocco, utilizzando la *virtual network* ordinata; svuota la lista degli sharers e transisce nello stato P.

Si supponga che la L2-X riceva la `GETS/GET_INSTR` inoltratagli dalla Dir$ prima di ricevere il `DATA_EXCLUSIVE` proveniente dalla Directory. Per cui la L2-X deve aggiungere l'identificatore del richiedente in una propria lista degli sharers; inoltre essa invia `GETS/GET_INSTR` verso la L1-X per poi transire nello stato IES. Per ogni altra richiesta `GETS/GET_INSTR` proveniente dalla Dir$, la L2-X deve solo memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato. La L1-X vedendosi recapitare una `GETS/GET_INSTR`, non fa altro che transitre anch'essa in IES.

Si supponga anche che la L2-X riceva la `GETX`, che contiene il numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso, inoltratagli dalla Dir$ prima di ricevere il `DATA_EXCLUSIVE` proveniente dalla Directory. Per cui la L2-X deve memorizzare l'identificatore del richiedente nel campo `Exclusive` assieme al numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso (tale numero viene effettivamente memorizzato nel campo `ForwardGetX_AckCount` della TBE); inoltre essa invia `GETX` verso la L1-X per poi transire nello stato IEI. La L1-X, vedendosi recapitare una `GETX`, non fa altro che transire anch'essa in IEI.

Quando arriva il `DATA_EXCLUSIVE` proveniente dalla Directory, la L2-X inoltra la copia del blocco di memoria in esso contenuta alla L1-X per mezzo di un messaggio `DATA_EXCLUSIVE` e confeziona un nuovo messaggio di tipo `EJECT`, contenente anche il proprio identificatore, da inviare verso la Dir$. La L1-X, in modo analogo, si vede recapitare `DATA_EXCLUSIVE` e, dopo aver memorizzato la copia del blocco di memoria in esso contenuta e, sbloccato il processore per permettere la Load Hit, invia anch'essa un messaggio di tipo `EJECT` verso la L2-X.

Non appena arriva il messaggio `EJECT`, la Dir$ confeziona ed invia un messaggio `WB_ACK` verso la L2-X utilizzando la *virtual network* ordinata, anche se essa non è l'attuale proprietaria del blocco.

Non appena arriva il messaggio EJECT proveniente dalla L1-X, la L2-X le risponde inviandole il WB_ACK; invece, non appena arriva il WB_ACK dalla Dir$, la L2-X invia la copia del blocco di memoria, incapusulato nel messaggio DATA_SHARED, ai banchi di L2$ il cui identificatore è contenuto nella lista degli sharers, svuotando infine tale lista. La copia del blocco di memoria va anche inviato, incapsulato sempre nel messaggio DATA_SHARED, alla L2-Z il cui identificatore è memorizzato nel campo Exclusive assieme al numero dei banchi di L2$ che in precedenza hanno richiesto il blocco in modo non esclusivo. Infine la L2-X transisce nello stato I. Analogamente la L1-X, vedendosi recapitare il WB_ACK dalla L2-X, transisce anch'essa nello stato I.

Ogni banco di L2$ che aveva richiesto il blocco di memoria in modo non esclusivo, a seguito della ricezione del messaggio INV, contenente l'identificatore della L2-Z che ha richiesto il blocco di memoria in modo esclusivo, confeziona ed invia a quest'ultima un messaggio di tipo ACK e poi transisce nello stato IEI; quando poi riceve la copia del blocco di memoria, per mezzo del DATA_SHARED, invia a sua volta un messaggio EJECT alla Dir$ che risponderà con un messaggio WB_ACK che scatenerà, a sua volta, la transizione nello stato I. La L2$ che aveva richiesto la copia del blocco di memoria in modo esclusivo, all'arrivo dei vari ACK e DATA_SHARED, transirà nello stato M come descritto in §4.4.2.

Si noti:

- che il messaggio EJECT proveniente dalla L1-X potrebbe arrivare alla L2-X dopo l'arrivo del WB_ACK proveniente dalla Dir$, in tal caso bisogna prevedere questo evento anche quando una L2$ si trova nello stato I;

- che la richiesta GETX, proveniente dalla Dir$, impedisce che alla L2-X arrivino altre richieste GETS/GET_INSTR, poichè viaggiano tutte su una *virtual network* ordinata e la GETX impone un cambiamento del proprietario del blocco nella Directory;

- che la richiesta GETX, proveniente dalla Dir$, potrebbe arrivare alla L2-X a seguito del messaggio DATA_EXCLUSIVE: in tal caso, la transizione transizione nello stato IEI avverrebbe dopo l'invio della ACCEPTS alla Dir$, non alterando di fatto la semantica del protocollo.

4.5.4 Store multiple

La Figura 29 illustra lo scenario in cui due processori, X ed Y, necessitano effettuare operazioni di Store in modo concorrente su un blocco di memoria non condiviso ed appartenente alla Directory.

La L1-X invia `GETX` alla L2-X e transisce nello stato IM. La L2-X analogamente invia `GETX`, contenente anche il proprio identificatore, alla Dir$ e transisce anch'essa nello stato IM. La Dir$, essendo proprietaria del blocco di memoria non condiviso, confeziona ed invia `GETX`, che contiene di identificatore del richiedente, verso la Directory; inoltre imposta l'`Owner` con l'identificatore del banco della L2$ richiedente e transisce nello stato P. La Directory, ricevendo `GETX` proveniente dalla DirectoryCache, preleva il blocco dalla memoria principale, lo incapsula nel messaggio `DATA_EXCLUSIVE`, e lo invia alla L2-X richiedente senza transire di stato.

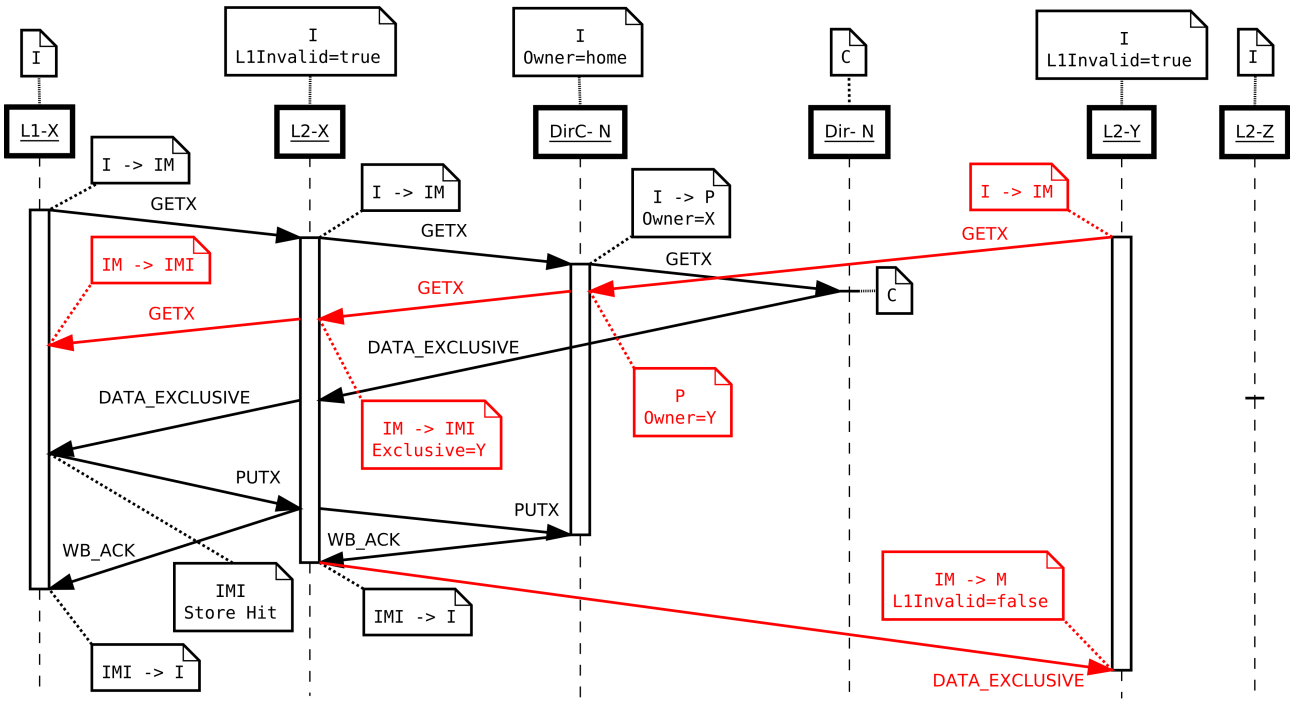

Figura 29 Store concorrenti con L2 Miss

Si supponga che, prima che la L2-X riceva il `DATA_EXCLUSIVE` proveniente dalla Directory, alla Dir$ arrivi una richiesta `GETX` per il blocco in oggetto. La Dir$ inoltra tale richiesta verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata ed imposta l'`Owner` con l'identificatore della nuova L2Cache richiedente (Y), non modificando lo stato.

Si supponga anche che la L2-X riceva la GETX inoltratagli dalla Dir$ prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve memorizzare l'identificatore del richiedente nel campo Exclusive; inoltre essa invia GETX verso la L1-X per poi transire IMI poichè un'altra L2Cache richiede lo stesso blocco di memoria in modo esclusivo. La L1-X, vedendosi recapitare una GETX, non fa altro che transire anch'essa in IMI.

Quando arriva il DATA_EXCLUSIVE proveniente dalla Directory, la L2-X inoltra il blocco di memoria in esso contenuto alla L1-X per mezzo di un messaggio DATA_EXCLUSIVE senza transire di stato. La L1-X si vede recapitare DATA_EXCLUSIVE e, dopo aver memorizzato il blocco in esso contenuto e sbloccato il processore per permettere la Store Hit, invia un messaggio di tipo PUTX verso la L2-X che contiene la copia aggiornata del blocco di memoria a seguito della Store.

Non appena arriva il messaggio PUTX dalla L1-X, la L2-X memorizza la copia del blocco di memoria, confeziona ed invia un messaggio WB_ACK verso la L1-X, ed invia la copia aggiornata del blocco di memoria, incapsulato in un messaggio di tipo PUTX, verso la Dir$ che, una volta ricevutolo, risponde al mittente con un WB_ACK, utilizzando la *virtual network* ordianata. La Dir$ ingnora l copia del blocco di memoria inviatole dalla L2-X in quanto quest'ultima non è più la proprietaria del blocco in oggetto.

Non appena arriva il WB_ACK dalla Dir$, la L2-X invia la copia aggiornata del blocco di memoria, incapusulato nel messaggio DATA_EXCLUSIVE, alla L2-Y (il cui identificatore è contenuto nel campo Exclusive), finendo per transire nello stato I. Analogamente la L1-X, vedendosi recapitare il WB_ACK dalla L2-X, transisce anch'essa nello stato I mentre la L2-Y richiedente il blocco escusivo passerà nello stato M non appena riceverà il messaggio DATA_EXCLUSIVE inviatogli dalla L2-X come descritto in §4.4.2.

4.5.5 Store e Load oppure iFetch

La Figura 30 illustra lo scenario in cui un processore necessita effettuare un operazione di Store in modo concorrente ad un'altro che necessita effettuare Load oppure iFetch su un blocco di memoria non condiviso ed appartenente alla Directory.

La L1-X invia GETX alla L2-X e transisce nello stato IM. La L2-X analogamente invia GETX, contenente il proprio identificatore, alla Dir$ e transisce anch'essa nello stato IM. La Dir$, essendo proprietaria del blocco di memoria non condiviso, confeziona ed invia una richiesta GETX, contenente l'identificatore del richiedente, verso la Directory; inoltre imposta l'Owner con l'identificatore della L2-X richiedente e transisce nello stato P. La Directory, ricevendo una richiesta GETX proveniente dalla DirectoryCache, preleva il blocco dalla memoria principale, lo incapsula nel messaggio DATA_EXCLUSIVE, e lo invia alla L2-X richiedente senza transire di stato.

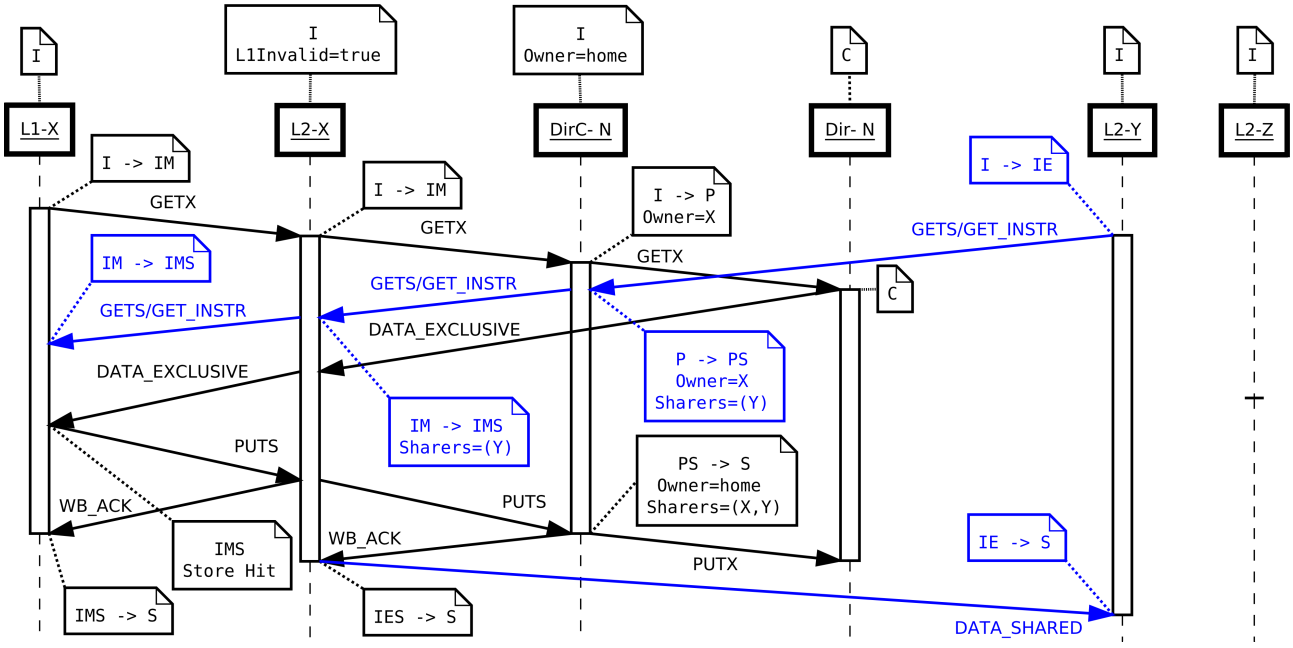

Figura 30 Store e Load oppure iFetch

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco in oggetto. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata alla L2-X proprietaria del blocco, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente il blocco alla lista degli sharers e transisce nello stato PS.

Si supponga inoltre che la L2-X riceva la GETS/GET_INSTR inoltratagli dalla Dir$ prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve aggiungere l'identificatore del richiedente nella propria lista degli sharers; inoltre essa invia GETS/GET_INSTR verso la L1-X per poi transire nello stato IMS: Invalid to Modified to Shared. Per ogni altra richiesta GETS/GET_INSTR proveniente dalla Dir$, la L2-X deve solo

memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato. L1-X vedendosi recapitare una GETS/GET_INSTR, non fa altro che transire anch'essa in IMS.

Quando arriva il DATA_EXCLUSIVE proveniente dalla Directory, la L2-X inoltra il blocco di memoria in esso contenuto alla L1-X per mezzo di un messaggio DATA_EXCLUSIVE. La L1-X si vede recapitare DATA_EXCLUSIVE e, dopo aver memorizzato il blocco in esso contenuto e sbloccato il processore per permettere la Store Hit, invia a un messaggio di tipo PUTS verso la L2-X che contiene la copia aggiornata del blocco di memoria.

Non appena arriva il messaggio PUTS dalla L1-X, la L2-X confeziona ed invia un messaggio WB_ACK verso la L1-X; memorizza la copia aggiornata del blocco di memoria e lo invia incapsulato in un messaggio di tipo PUTS, contenente anche il proprio identificatore, verso la Dir$. Quest'ultima, una volta ricevutolo, risponde al mittente con un WB_ACK, utilizzando la *virtual network* ordinata; invia la copia aggirnata blocco di memoria alla Directory mediante PUTX sulla *virtual network* ordinata; aggiunge l'identificatore del banco della L2Cache contenuto nel campo Owner alla lista degli sharers, diventa proprietaria del blocco assegnando il valore home al campo Owner; ed infine transisce nello stato S. La Directory, invece, vedendosi recapitare un blocco di memoria per mezzo di una PUTX, non fa altro che memorizzarlo nella memoria principale.

La L1-X attende l'arrivo del WB_ACK proveniente dalla L2-X per transire nello stato S. La L2-X, una volta ricevuto il WB_ACK proveniente dalla Dir$, invia la copia aggiornata del blocco di memoria, per mezzo di messaggi DATA_SHARED, ad ogni banco di L2$ il cui identificatore è contenuto nella lista degli sharers; svuota tale lista e transisce nello stato S. Ogni banco di L2Cache che aveva richiesto il blocco in modo non esclusivo, una volta ricevuto il DATA_SHARED, transisce in S come descritto in §4.4.1.

Si noti:

- che l'arrivo di ulteriori GETS/GET_INSTR dalla Dir$ sono possibili anche a seguito della recezione del DATA_EXCLUSIVE proveniente dalla Directory oppure a seguito della ricezione della PUTS proveniente dalla L1-X.

- che l'arrivo del `WB_ACK` impedisce che alla L2-X arrivino altre richieste `GETS/GET_INSTR` poichè viaggiano tutte su una *virtual network* ordinata e sono tutte richieste precedenti al `WB_ACK` che rende la Directory proprietaria del blocco.

4.5.6 Store, Load oppure iFetch multiple e Store

La Figura 31 illustra lo scenario in cui un processore necessita effettuare un operazione di Store in modo concorrente ad altri processori che necessitano effettuare operazioni di Load oppure iFetch su un di memoria non condiviso, appartenente alla Directory; in cui vi è anche un altro processore ancora che necessita di effettuare un'ulteriore operazione di Store sullo stesso blocco.

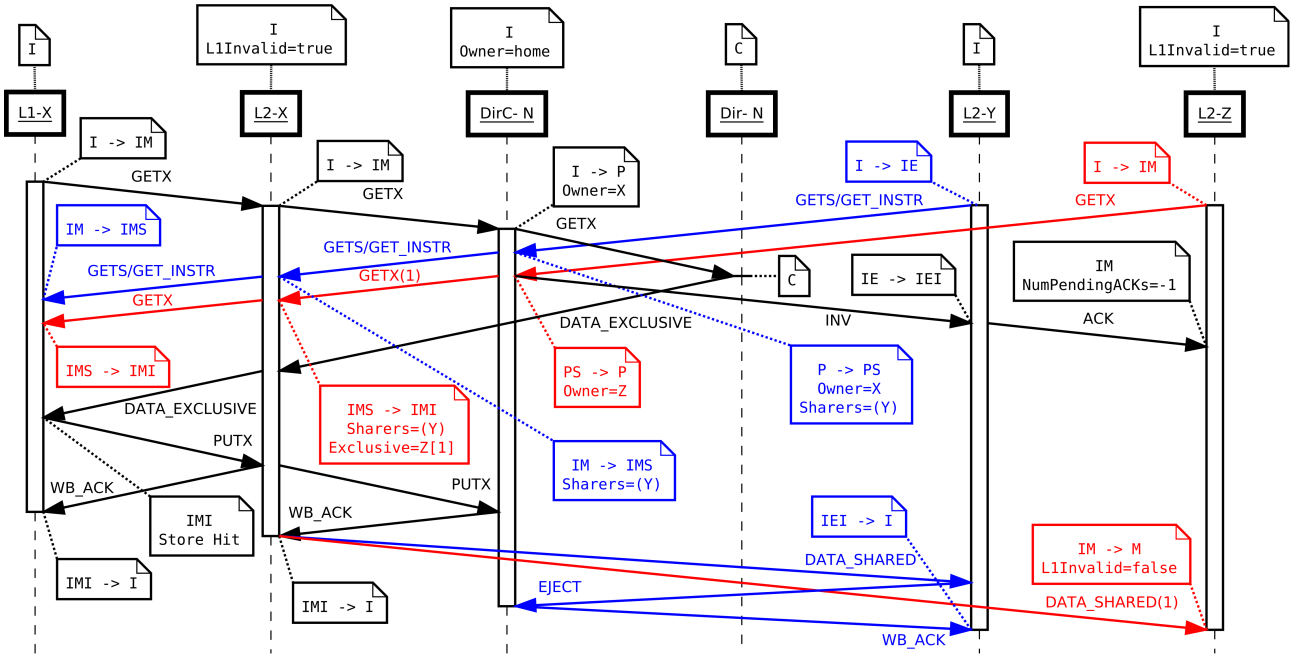

Figura 31 Store, Load oppure iFetch multiple e Store

La L1-X invia `GETX` alla L2-X e transisce nello stato IM. La L2-X analogamente invia `GETX`, contenente il proprio identificatore, alla Dir$ e transisce anch'essa nello stato IM. La Dir$, essendo proprietaria del blocco di memoria non condiviso, confeziona ed invia `GETX`, contenente l'identificatore del richiedente, verso la Directory; inoltre imposta l'`Owner` con l'identificatore della L2-X e transisce nello stato P. La Directory, ricevendo una `GETX` proveniente dalla Dir$, preleva il blocco dalla memoria principale, lo incapsula nel messaggio `DATA_EXCLUSIVE`, e lo invia alla L2-X richiedente senza transire di stato.

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco in oggetto. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata alla L2-X proprietaria del blocco di memoria, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente alla lista degli sharers e transisce nello stato PS.

Si supponga che, prima che la L2-X riceva il DATA_EXCLUSIVE proveniente dalla Directory, alla Dir$ arrivi una richiesta GETX per il blocco in oggetto. La Dir$ inoltra tale richiesta, assieme al numero degli identificatori dei banchi della L2$ presenti nella lista degli sharers, verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata; imposta l'Owner con l'identificatore del banco della L2$ richiedente (Z); invia ad ogni banco di L2$, il cui identificatore è presente nella lista degli sharers, un messaggio di tipo INV che contiene al suo interno anche l'identificatore del nuovo proprietario del blocco utilizzando la *virtual network* ordinata; svuota la lista degli sharers e transisce nello stato P.

Si supponga che la L2-X riceva la GETS/GET_INSTR inoltratagli dalla Dir$ prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve aggiungere l'identificatore del richiedente nella propria lista degli sharers; inoltre essa invia GETS/GET_INSTR verso la L1-X per poi transire nello stato IMS. Per ogni altra richiesta GETS/GET_INSTR proveniente dalla DirectoryCache, la L2-X deve solo memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato. L1-X vedendosi recapitare una GETS/GET_INSTR, non fa altro che transire anch'essa in IMS.

Si supponga che la L2-X riceva la GETX, assieme al numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso, inoltratagli dalla Dir$, prima di ricevere il DATA_EXCLUSIVE proveniente dalla Directory. Per cui la L2-X deve memorizzare l'identificatore del richiedente nel campo Exclusive assieme al numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso; inoltre essa invia GETX verso la L1-X per poi transire nello stato IMI. La L1-X, vedendosi recapitare una GETX, non fa altro che transire anch'essa in IMI.

Quando arriva il DATA_EXCLUSIVE proveniente dalla Directory, la L2-X inoltra la copia del blocco di memoira in esso contenuto alla L1-X per mezzo di un messaggio DATA_EXCLUSIVE. La L1-X si vede recapitare DATA_EXCLUSIVE e, dopo aver memorizzato il blocco in esso contenuto

e sbloccato il processore per permettere la Store Hit, invia a un messaggio di tipo PUTX verso la L2-X che contiene la copia aggiornata del blocco di memoria modificato dalla Store.

Non appena arriva il messaggio PUTX dalla L1-X, la L2-X confeziona ed invia un messaggio WB_ACK verso la L1-X; memorizza ed invia il blocco ricevuto in un messaggio di tipo PUTX, contenente anche il proprio identificatore, verso la Dir$; non transisce di stato. La Dir$, una volta ricevuta la PUTX dalla L2-X, risponde al mittente con un WB_ACK sulla *virtual network* ordinata; ignora il blocco di memoria ricevuto, in quanto il mittente non è più l'attuale proprietario.

La L1-X attende l'arrivo del WB_ACK proveniente dalla L2-X per transire nello stato I.

La L2-X, una volta ricevuto il WB_ACK proveniente dalla Dir$, invia la copia aggiornata del blocco di memoria, per mezzo di messaggi DATA_SHARED, ad ogni banco di L2$ il cui identificatore è contenuto nella lista degli sharers, svuotando la lista; invia anche la copia aggiornata del blocco, per mezzo di un messaggio DATA_SHARED contente anche il numero dei banchi di L2$ che hanno richiesto il blocco in modo non sesclusivo, alla L2-Z che ha rischiesto il blocco in modo esclusivo; transisce nello stato I.

Ogni banco di L2$ che aveva richiesto una copia del blocco di memoria in modo non esclusivo, a seguito della ricezione del messaggio INV contenente l'identificatore del banco della L2$ che ha richiesto la copia in modo esclusivo, confeziona ed invia a quest'ultima un messaggio di tipo ACK e poi transisce nello stato IEI; quando poi riceve la copia del blocco, per mezzo del DATA_SHARED, invia a sua volta un messaggio EJECT alla Dir$ che risponderà con un messaggio WB_ACK che scatenerà, a sua volta, la transizione nello stato I. La L2$ che aveva richiesto la copia del blocco di memoria in modo esclusivo, all'arrivo dei vari ACK e DATA_SHARED, transirà nello stato M come descritto in §4.4.2.

Si noti:

- che la richiesta GETX, proveniente dalla Dir$, impedisce che alla L2-X arrivino altre richieste GETS/GET_INSTR poichè esse viaggiano tutte su una *virtual network* ordinata e sono tutte precedenti alla GETX, che impone un cambiamento del proprietario nella Directory;

- che la richiesta GETX, proveniente dalla Dir$, potrebbe arrivare alla L2-X a seguito del messaggio DATA_EXCLUSIVE: in tal caso, la transizione nello stato IMI avverrebbe dopo l'invio della PUTS alla DirectoryCache, non alterando di fatto la semantica del protocollo.

4.6 Operazioni remote

Si supponga che per un dato processore X, la copia di un determinato blocco di memoria sia presente in un banco della L2-X, ed eventualmente nella L1-X. Si andrà ora ad illustrare come si comporta il protocollo di coerenza quando più processori (non X) necessitano effettuare operazioni sul blocco di memoria in oggetto.

4.6.1 Load oppure iFetch multiple

La Figura 32 illustra lo scenario in cui il processore Y necessita di effettuare l'operazione di Load oppure iFetch su un blocco di memoria presente in modo esclusivo nella L2-X, ma non presente nella L1-X.

La L1-X si trova nello stato I; La L2-X si trova indifferentemente nello stato E oppure M; la Dir$ possiede il blocco di directory nello stato P con il campo Owner impostato a X; la Directory è nello stato C e la L2-Y nello stato I.

Si supponga che alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco di memoria la cui copia è posseduta dalla L2-X. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata al banco della L2$ proprietaria del blocco di memoria, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente alla lista degli sharers e transisce nello stato PS.

La L2-X, a seguito della ricezione della GETS/GET_INSTR proveniente dalla DirectoryCache, invia la copia del blocco di memoria, incapsulato nel messaggio DATA_SHARED, al banco di L2$ richiedente e, invia alla Dir$ il messaggio ACCEPTS, se si trova nello stato E, oppure il messaggio PUTS contenente la copia del blocco di memoria, se si trova nello stato M. Transisce, infine, nello stato MS: Modified to Shared. Per ogni altra richiesta di tipo GETS/GET_INSTR proveniente dalla Dir$, la L2-X invia la copia del blocco di memoria, incapsulato nel messaggio DATA_SHARED, al banco di L2$ richiedente senza transire di stato.

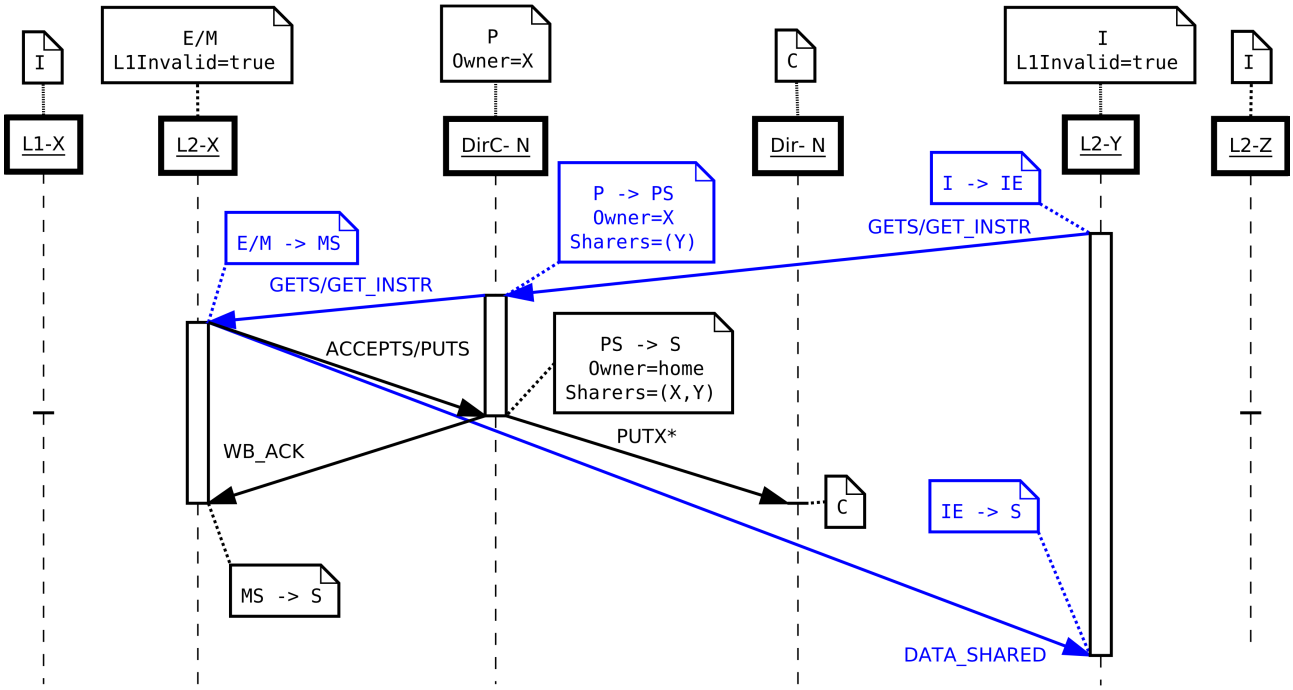

Figura 32 Load oppure iFetch multiple (L1 Miss)

La Dir$, nel caso in cui dovesse ricevere la PUTS dalla L2-X proprietaria del blocco, inoltra la copia del blocco in esso contenuta, per mezzo del messaggio PUTX sulla *virtual network* ordinata, alla Directory che provvederà alla memorizzazione in memoria principale. In ogni caso, anche quello in cui la Dir$ dovesse ricevere ACCEPTS, aggiunge alla lista degli sharers l'identificatore della L2-X proprietaria del blocco contenuto nel campo Owner; imposta quest'ultimo ad home diventando di fatto proprietaria del blocco; invia il messaggio WB_ACK, utilizzando la *virtual network* ordinata, alla L2-X che le aveva inviato la ACCEPTS/PUTS; ed infine transisce nello stato S.

La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce in S; mentre ciascun banco di L2$ che riceve DATA_SHARED transisce in S.

La Figura 33 illustra lo scenario precedente con la differenza che ora il blocco in oggetto è anche memorizzato nella L1-X.

La L1-X si trova indifferentemente nello stato E oppure M; La L2-X si trova indifferentemente nello stato E oppure M; la Dir$ possiede il blocco di directory nello stato P con il campo Owner impostato a X; la Directory è nello stato C e la L2-Y nello stato I.

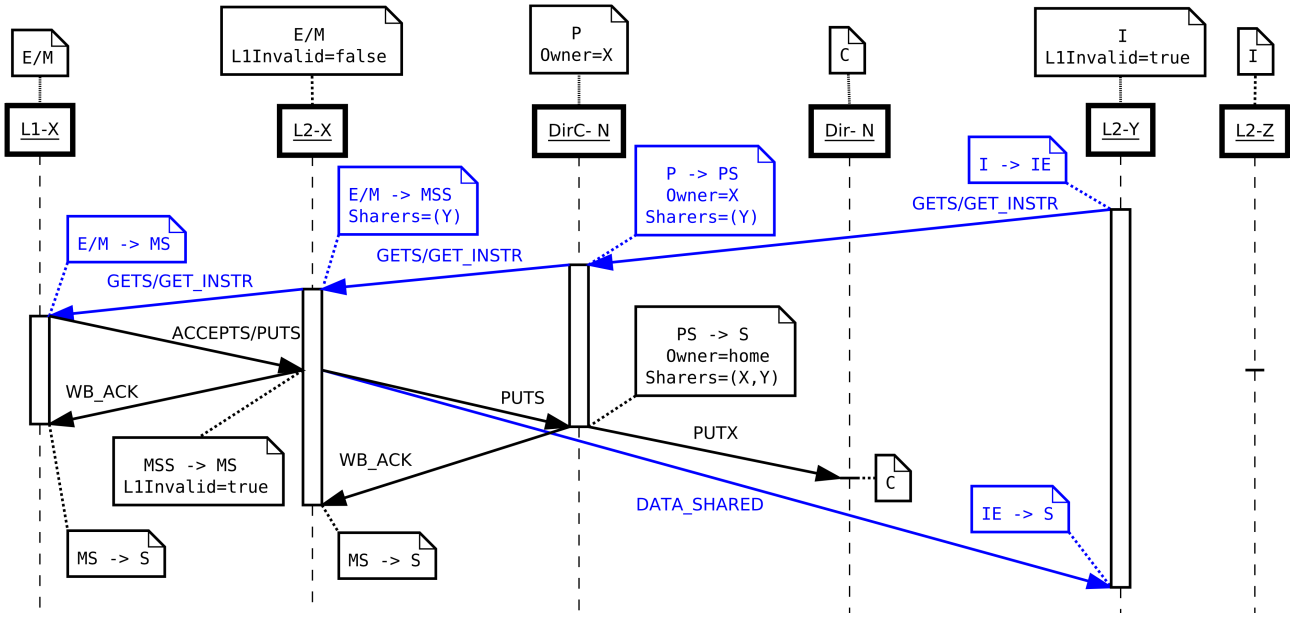

Figura 33 Load oppure iFetch multiple (L1 Hit)

Si supponga che alla Dir$ arrivino una o più richieste `GETS/GET_INSTR` per il blocco di memoria posseduto dalla L2-X. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata alla L2-X proprietaria del blocco di memoria, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente alla lista degli sharers e transisce nello stato PS.

La L2-X, ricevuta la `GETS/GET_INSTR` inoltratagli dalla Dir$, aggiunge l'identificatore del richiedente nella propria lista degli sharers; invia `GETS/GET_INSTR` verso la L1-X per poi transire nello stato MSS: Modified to Shared waiting for L1 Sharing. Per ogni altra richiesta `GETS/GET_INSTR` proveniente dalla Dir$, la L2-X deve solo memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato.

La L1-X, ricevuta la `GETS/GET_INSTR` proveniente dalla L2-X, invia a quest'ultima il messaggio `ACCEPTS`, se si trova nello stato E, oppure il messaggio `PUTS` contenente la copia del blocco di memoria, se si trova nello stato M. Transisce, infine, nello stato MS.

La L2-X, a seguito della ricezione della `ACCEPTS/PUTS` proveniente dalla L1-X, risponde a quest'ultima con un `WB_ACK`; memorizza il blocco di memoria eventualmente ricevuto; invia il blocco di memoria incapsulato nel messaggio `DATA_SHARED` ad ogni banco di L2$ contenuto

nella lista degli sharers; svuota tale lista; invia alla Dir$ il messaggio `PUTS` contenente il blocco di memoria; imposta il flag `L1Invalid` a `true`; transisce nello stato MS. Per ogni altra richiesta di tipo `GETS/GET_INSTR` proveniente dalla Dir$, la L2-X invia la copia del blocco di memoria, incapsulato nel messaggio `DATA_SHARED`, al banco di L2$ richiedente senza transire di stato.

La Dir$, ricevuta la `PUTS` contenente il blocco di memoria proveniente dalla L2-X proprietaria del blocco, inoltra il blocco stesso, per mezzo del messaggio `PUTX`, alla Directory, la quale provvederà alla memorizzazione in memoria principale. La Dir$, inoltre, aggiunge alla lista degli sharers l'identificatore della L2-X proprietaria del blocco contenuto nel campo `Owner`; imposta quest'ultimo ad `home` diventando di fatto proprietaria del blocco; invia il messaggio `WB_ACK`, utilizzando la *virtual network* ordinata, alla L2-X ed infine transisce nello stato S.

La L1-X, ricevuto il WB_ACK proveniente dalla L2-X, transisce nello Stato S. Analogamente la L2-X, ricevuto il `WB_ACK` dalla Dir$, transisce nello stato S.

4.6.2 Load oppure iFetch multiple e Store

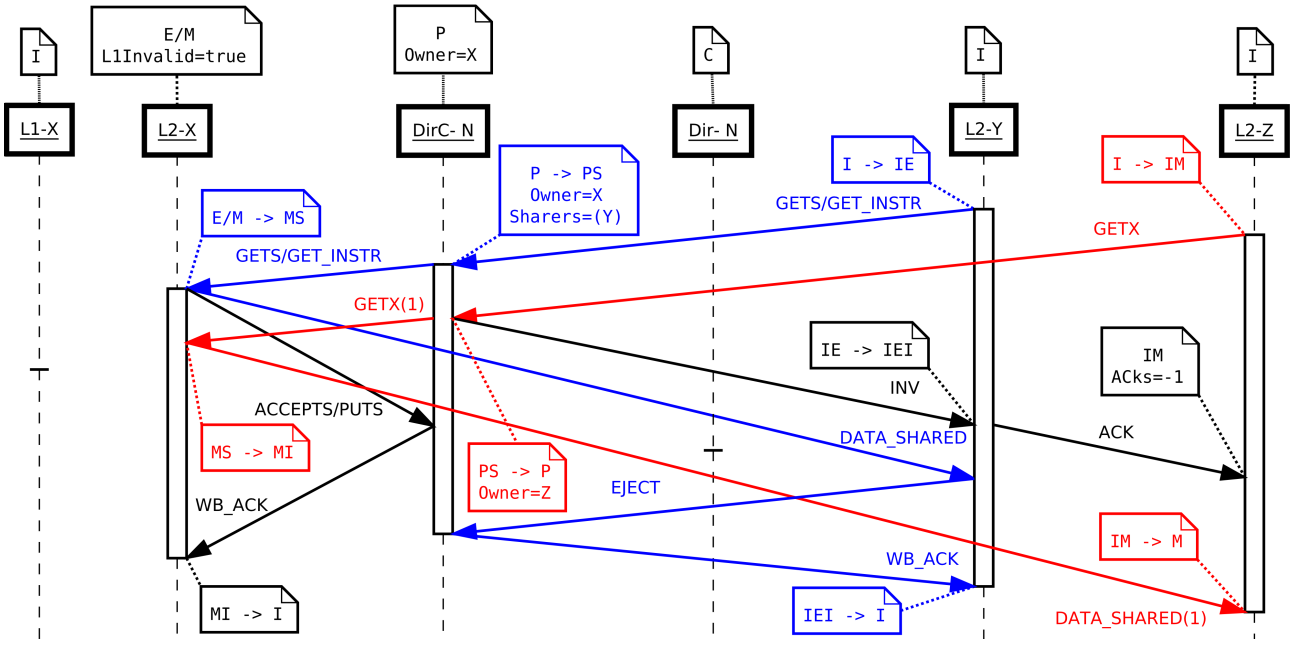

Figura 34 Load oppure iFetch multiple e Store (L1 Miss)

La Figura 34 illustra lo scenario in cui il processore Y necessita di effettuare l'operazione di Load oppure iFetch su un blocco di memoria presente in modo esclusivo nella L2-X, ma non presente

nella L1-X; inoltre il proccessore Z necessita di effettuare l'operazione di Store sullo stesso blocco.

La L1-X si trova nello stato I; La L2-X si trova indifferentemente nello stato E oppure M; la Dir$ possiede il blocco di directory nello stato P con il campo `Owner` impostato a X; la Directory è nello stato C; la L2-Y nello stato I; così come la L2-Z.

Si supponga che alla Dir$ arrivino una o più richieste `GETS/GET_INSTR` per il blocco di memoria la cui copia è posseduta dalla L2-X. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata al banco della L2$ proprietaria del blocco di memoria, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente alla lista degli sharers e transisce nello stato PS.

La L2-X, a seguito della ricezione della `GETS/GET_INSTR` proveniente dalla DirectoryCache, invia la copia del blocco di memoria, incapsulato nel messaggio `DATA_SHARED`, al banco di L2$ richiedente e, invia alla Dir$ il messaggio `ACCEPTS`, se si trova nello stato E, oppure il messaggio `PUTS` contenente la copia del blocco di memoria, se si trova nello stato M. Transisce, infine, nello stato MS. Per ogni altra richiesta di tipo `GETS/GET_INSTR` proveniente dalla Dir$, la L2-X invia la copia del blocco di memoria, incapsulato nel messaggio `DATA_SHARED`, al banco di L2$ richiedente senza transire di stato.

Si supponga che, prima di ricevere `ACCEPTS/PUTS` proveniente dalla L2-X, alla Dir$ arrivi una richiesta `GETX` per il blocco in oggetto. La Dir$ inoltra tale richiesta, assieme al numero degli identificatori dei banchi della L2$ presenti nella lista degli sharers, verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata; imposta l'`Owner` con l'identificatore del banco della L2$ richiedente (Z); invia ad ogni banco di L2$, il cui identificatore è presente nella lista degli sharers, un messaggio di tipo `INV` che contiene al suo interno anche l'identificatore del nuovo proprietario del blocco utilizzando la *virtual network* ordinata; svuota la lista degli sharers e transisce nello stato P.

La L2-X, a seguito della ricezione della `GETX` proveniente dalla Dir$, invia la copia del blocco di memoria, per mezzo di un messaggio `DATA_SHARED` contente anche il numero dei banchi di L2$ che hanno richiesto il blocco in modo non sesclusivo, alla L2-Z che ha rischiesto il blocco in modo esclusivo e transisce nello stato MI.

La Dir$, ricevuto ACCEPTS/PUTS dalla L2-X, risponde al mittente con il WB_ACK sulla *virtual nertwork* ordinata; senza transire di stato ed ignorando l'eventuale blocco di memoria ricevuto poichè ormai la L2-X non è più proprietaria del blocco. La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce in I.

Ogni banco di L2$ che aveva richiesto una copia del blocco di memoria in modo non esclusivo, a seguito della ricezione del messaggio INV contenente l'identificatore del banco della L2$ che ha richiesto la copia in modo esclusivo, confeziona ed invia a quest'ultima un messaggio di tipo ACK e poi transisce nello stato IEI; quando poi riceve la copia del blocco, per mezzo del DATA_SHARED, invia a sua volta un messaggio EJECT alla Dir$ che risponderà con un messaggio WB_ACK che scatenerà, a sua volta, la transizione nello stato I. La L2Cache che aveva richiesto la copia del blocco di memoria in modo esclusivo, all'arrivo dei vari ACK e DATA_SHARED, transirà nello stato M come descritto in §4.4.2.

La Figura 35 illustra lo scenario precedente con la differenza che adesso il blocco in oggetto è anche memorizzato nella L1-X.

La L1-X si trova indifferentemente nello stato E oppure M; La L2-X si trova indifferentemente nello stato E oppure M; la Dir$ possiede il blocco di directory nello stato P con il campo Owner impostato a X; la Directory è nello stato C; la L2-Y nello stato I; così come la L2-Z.

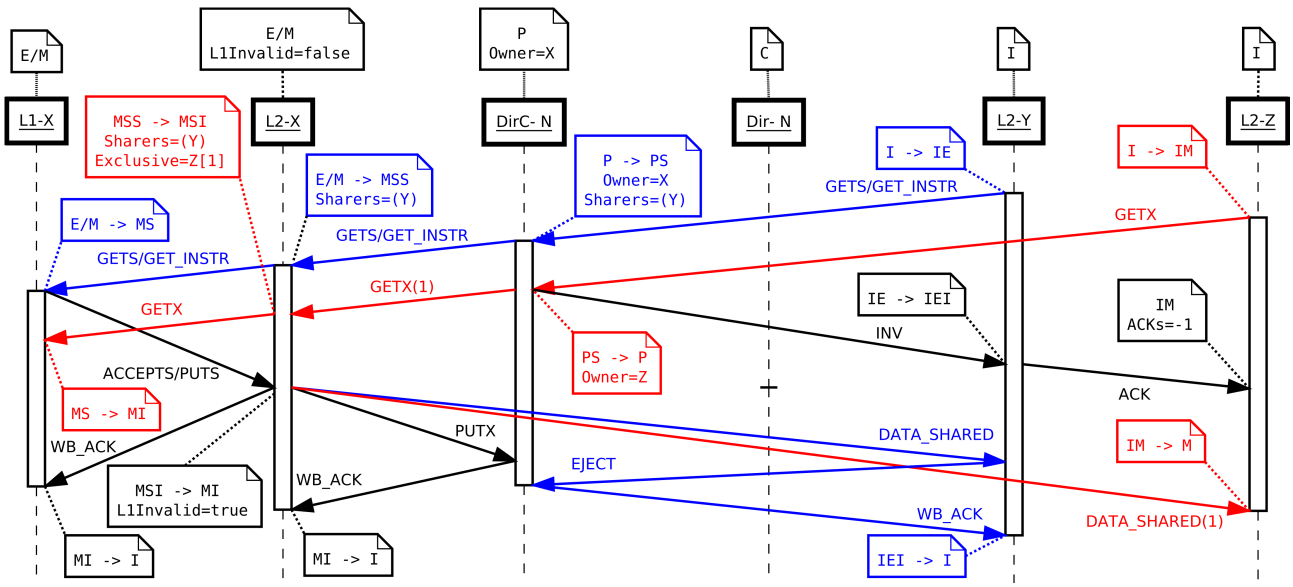

Figura 35 Load oppure iFetch multiple e Store (L1 Hit)

Si supponga che alla Dir$ arrivino una o più richieste GETS/GET_INSTR per il blocco di memoria la cui copia è posseduto dalla L2-X in modo esclusivo. La Dir$ inoltra tali richieste verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata. Per ogni richiesta ricevuta ed inoltrata al banco della L2$ proprietaria del blocco di memoria, la Dir$ aggiunge l'identificatore del banco della L2$ richiedente alla lista degli sharers e transisce nello stato PS.

La L2-X, ricevuta la GETS/GET_INSTR inoltratagli dalla Dir$, aggiunge l'identificatore del richiedente nella propria lista degli sharers; invia GETS/GET_INSTR verso la L1-X per poi transire nello stato MSS. Per ogni altra richiesta GETS/GET_INSTR proveniente dalla Dir$, la L2-X deve solo memorizzare l'identificatore del richiedente nella lista degli sharers senza inviare nulla alla L1-X e senza effettuare transizioni di stato.

La L1-X, ricevuta la GETS/GET_INSTR proveniente dalla L2-X, invia a quest'ultima il messaggio ACCEPTS, se si trova nello stato E, oppure il messaggio PUTS contenente la copia del blocco di memoria, se si trova nello stato M. Transisce, infine, nello stato MS.

Si supponga che, prima che la L2-X abbia ricevuto ACCEPTS/PUTS proveniente dalla L1-X, alla Dir$ arrivi una richiesta GETX per il blocco in oggetto. La Dir$ inoltra tale richiesta, assieme al numero degli identificatori dei banchi della L2$ presenti nella lista degli sharers, verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata; imposta l'Owner con l'identificatore del banco della L2$ richiedente (Z); invia ad ogni banco di L2$, il cui identificatore è presente nella lista degli sharers, un messaggio di tipo INV che contiene al suo interno anche l'identificatore del nuovo proprietario del blocco utilizzando la *virtual network* ordinata; svuota la lista degli sharers e transisce nello stato P.

Si supponga che la L2-X riceva la GETX, assieme al numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso, inoltratagli dalla Dir$, prima di ricevere ACCEPTS/PUTS proveniente dalla L1-X. Per cui la L2-X deve memorizzare l'identificatore del richiedente nel campo Exclusive assieme al numero di banchi di L2$ che hanno fatto richiesta del blocco condiviso; inoltre essa invia GETX verso la L1-X per poi transire nello stato MSI. La L1-X, vedendosi recapitare una GETX, non fa altro che transire in MI.

La L2-X, a seguito della ricezione della ACCEPTS/PUTS proveniente dalla L1-X, risponde a quest'ultima con un WB_ACK; memorizza il blocco di memoria eventualmente ricevuto; invia il

blocco di memoria incapsulato nel messaggio DATA_SHARED ad ogni banco di L2$ contenuto nella lista degli sharers; svuota tale lista; invia il blocco di memoria, per mezzo di un messaggio DATA_SHARED contente anche il numero dei banchi di L2$ che hanno richiesto il blocco in modo non sesclusivo, alla L2-Z che ha rischiesto il blocco in modo esclusivo; invia alla Dir$ il messaggio PUTX contenente il blocco di memoria; imposta il flag L1Invalid a true; transisce nello stato MS. La L1-X, ricevuto il WB_ACK proveniente dalla L2-X, transisce in I.

La Dir$, ricevuto PUTX dalla L2-X, risponde al mittente con il WB_ACK sulla *virtual nertwork* ordinata; senza transire di stato ed ignorando l'eventuale blocco di memoria ricevuto poichè ormai la L2-X non è più proprieratia del blocco. La L2-X, ricevuto il WB_ACK proveniente dalla Dir$, transisce in I.

Ogni banco di L2$ che aveva richiesto una copia del blocco di memoria in modo non esclusivo, a seguito della ricezione del messaggio INV contenente l'identificatore del banco della L2$ che ha richiesto la copia in modo esclusivo, confeziona ed invia a quest'ultima un messaggio di tipo ACK e poi transisce nello stato IEI; quando poi riceve la copia del blocco, per mezzo del DATA_SHARED, invia a sua volta un messaggio EJECT alla Dir$ che risponderà con un messaggio WB_ACK che scatenerà, a sua volta, la transizione nello stato I. La L2Cache che aveva richiesto la copia del blocco di memoria in modo esclusivo, all'arrivo dei vari ACK e DATA_SHARED, transirà nello stato M come descritto in §4.4.2.

4.6.3 Store

La Figura 36 illustra lo scenario in cui il processore Y necessita di effettuare l'operazione di Store su un blocco di memoria segnalato alla Directory come condiviso dalla L2-X, ed eventualmente dalla L1-X. Tale scenario è simile a quello già descritto in §4.4.2, ma adesso verranno messe in evidenza le interazioni tra la L1-X ed L2-X a seguito dell'arrivo del messaggio di invalidazione.

La Dir$ possiede il blocco di directory relativo al blocco di memoria in oggetto nello stato S con il campo Owner impostato ad home e l'identificatore di un banco della L2-X nella lista degli sharers. La Directory di conseguenza è nello stato C. I relativi blocchi di cache della L2-X e la L1-X si possono trovare nelle varie combinazioni di stato S/I che rispettano il principio di inclusione.

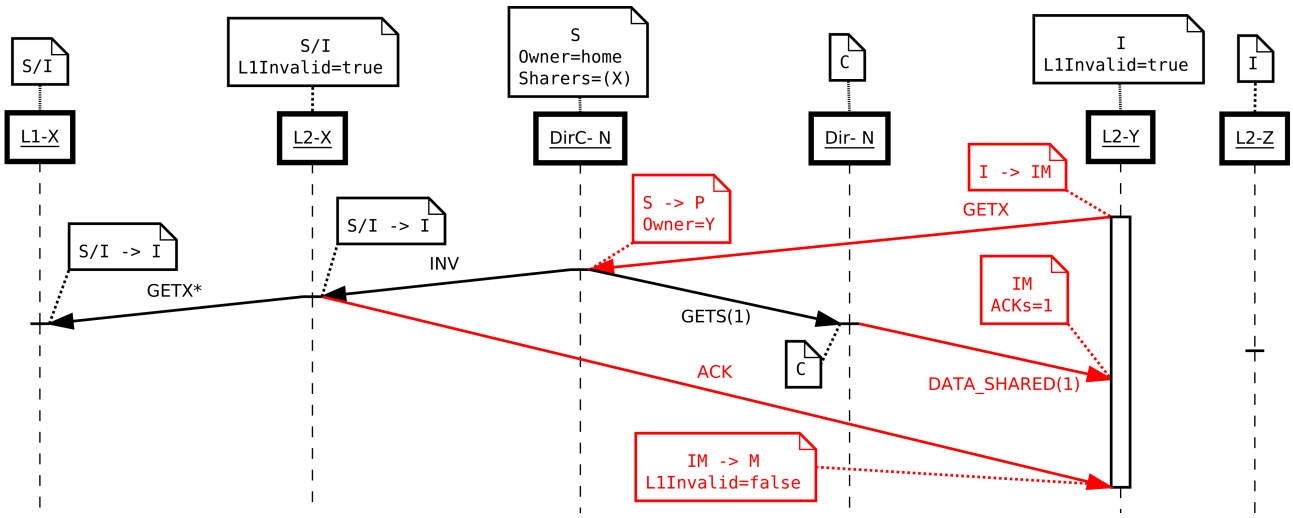

Figura 36 Store condivisa

Alla Dir$ arriva una richiesta `GETX` proveniente dalla L2-Y per il blocco di memoria in oggetto. La Dir$, come già descritto in §4.4.2, imposta l'`Owner` con l'identificatore del banco della L2$ richiedente (Y); invia ad ogni banco di L2$, il cui identificatore è presente nella lista degli sharers, un messaggio di tipo `INV`, contenente anche l'identificatore del nuovo proprietario del blocco, utilizzando la *virtual network* ordinata; svuota la lista degli sharers; invia un messaggio di richiesta del blocco di memoria condiviso (`GETS`) alla Directory utilizzando sempre la *virtual network* ordinata.Tale messaggio deve contenere anche il numero di acknowledgement (`ACK`) che il banco di L2-Y richiedente deve attendere prima di considerare invalidate le copie, del blocco di memoria che successivamente la Directory le invierà, eventualmente condivise dagli altri banchi della L2$. Infine transisce nello stato P.

La L2-X, che condivide (a meno di un rimpiazzamanto silente) il blocco di memoria, si vede recapitare un messaggio di invalidazione da parte della Dir$ e, come già descritto in §4.4.2, confeziona un messaggio di acknowledgement (`ACK`) e lo invia alla L2-Y che ha richiesto il bloccodi memoria come esclusivo; per poi transire nello stato I. Siccome per il principio di inclusione è possibile, a meno di rimpiazzamento silente, che se la L2-X condivide la copia blocco, anche la relativa L1-X la possa condividere. E' necessario quindi che la L2-X, a seguito della ricezione del messaggio di invalidazione, e prima di rispondere con l'acknowledgement e transire nello stato I, debba inviare un messaggio di tipo `GETX` alla L1-X in modo da provocarle un rimpiazzamento "forzato" (invalidazione).

La Directory, vedendosi recapitare il messaggio `GETS`, preleva il blocco dalla memoria principale, lo incapsula assieme al numero degli acknowledgement contenuti nella richiesta ricevuta, in un messaggio `DATA_SHARED` e, senza transire di stato, lo invia alla L2-Y richiedente, la quale conclude il protocollo come descritto in §4.4.2.

La Figura 37 illustra lo scenario in cui il processore Y necessita di effettuare l'operazione di Store su un blocco di memoria presente in modo esclusivo nella L2-X, ma non presente nella L1-X. La L1-X si trova nello stato I; La L2-X si trova indifferentemente nello stato E oppure M; la Dir$ possiede il blocco di directory nello stato P con il campo `Owner` impostato a X; la Directory è nello stato C; la L2-Y nello stato I; così come la L2-Z.

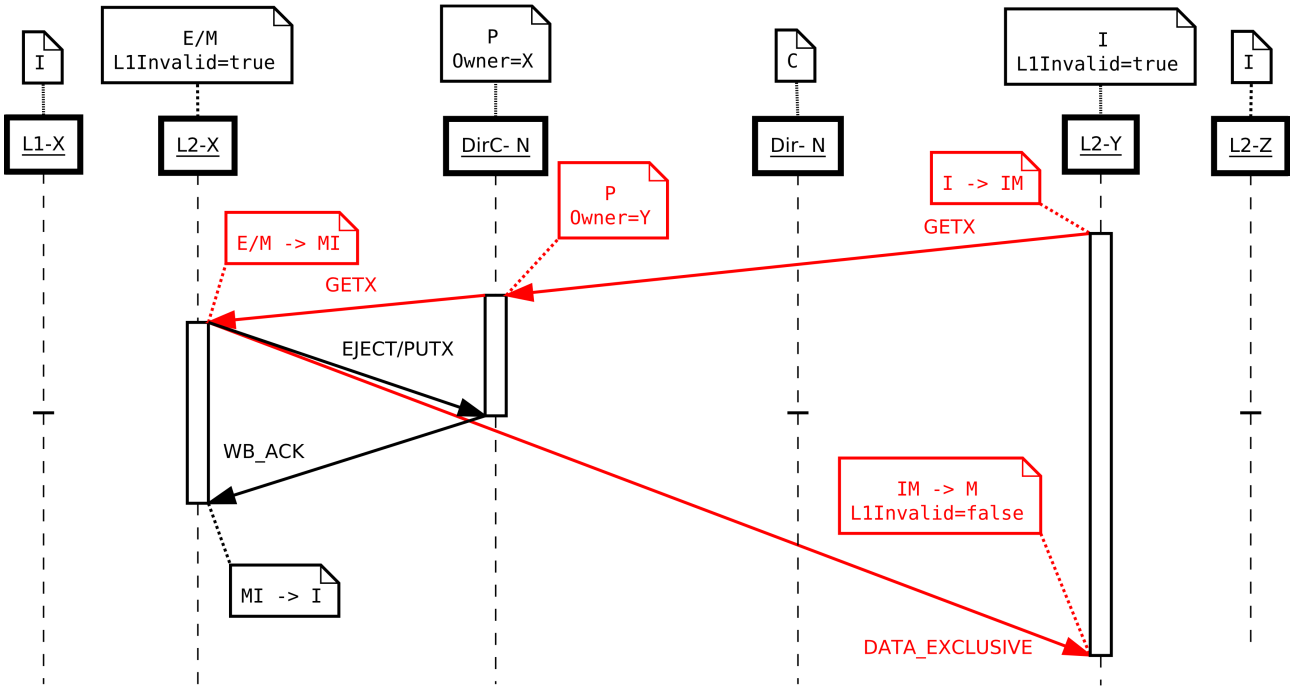

Figura 37 Store (L1 Miss)

Alla Dir$ arriva una `GETX` proveniente dalla L2-Y per il blocco di memoria in oggetto. La Dir$ inoltra la richiesta verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata ed imposta il campo `Owner` a Y senza transire di stato.

L'arrivo della `GETX` provoca nella L2-X un rimpiazzamento forzato: invia la copia del blocco di memoria, incapsulato nel messaggio `DATA_ECLUSIVE`, al banco di L2-Y richiedente e, invia alla Dir$ il messaggio `EJECT`, se si trova nello stato E, oppure il messaggio `PUTX` contenente la copia

del blocco di memoria modificato, se si trova nello stato M. Transisce, infine, nello stato MI.

La Dir$, ricevuto `EJECT/PUTX` dalla L2-X, risponde al mittente con il `WB_ACK` sulla *virtual nertwork* ordinata; senza transire di stato ed ignorando l'eventuale blocco di memoria ricevuto poichè ormai la L2-X non è più proprieratia del blocco.

La L2-Y, alla ricezione del `DATA_EXCLUSIVE`, conclude il protocollo come descritto in §4.4.2.

La Figura 38 illustra lo scenario precedente con la differenza che adesso il blocco in oggetto è anche memorizzato nella L1-X.

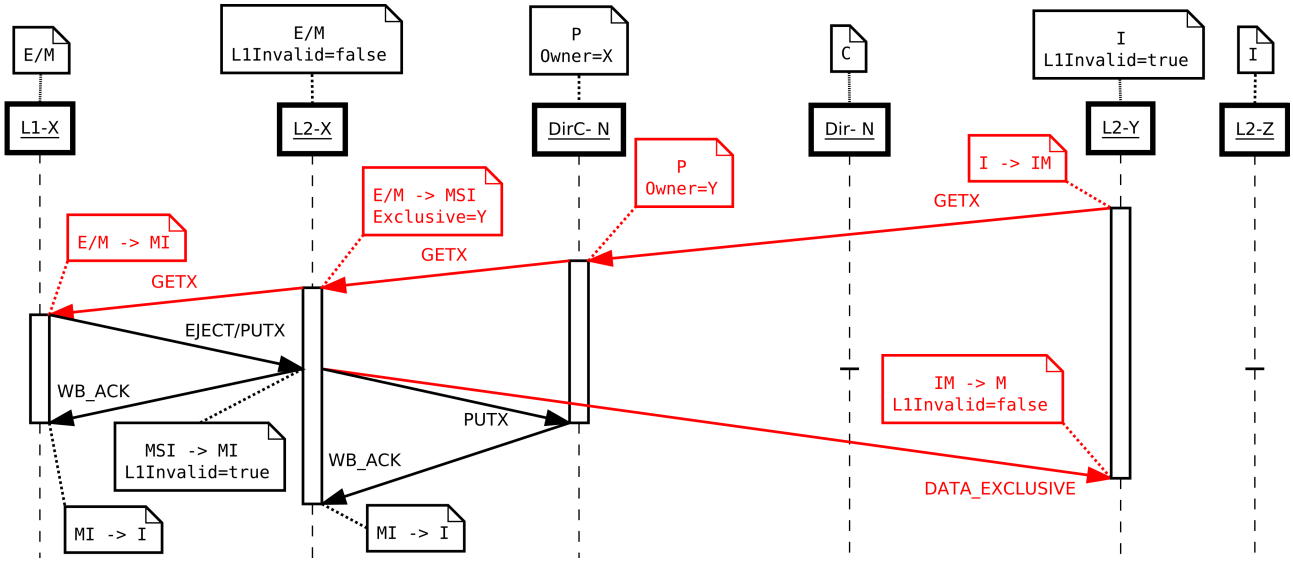

Figura 38 Store (L1 Hit)

La L1-X si trova indifferentemente nello stato E oppure M; La L2-X si trova indifferentemente nello stato E oppure M con il flag `L1Invalid` impostato a `false`; la Dir$ possiede il blocco di directory nello stato P con il campo `Owner` impostato a X; la Directory è nello stato C; la L2-Y nello stato I; così come la L2-Z.

Alla Dir$ arriva una `GETX` proveniente dalla L2-Y per il blocco di memoria in oggetto. La Dir$ inoltra la richiesta verso la L2-X proprietaria del blocco utilizzando la *virtual network* ordinata ed imposta il campo `Owner` a Y senza transire di stato.

L'arrivo della GETX provoca nella L2-X un rimpiazzamento forzato: invia un messaggio GETX verso la L1-X, memorizza l'identificatore del banco della L2$ richiedente (Y) nel campo Exclusive e transisce nello stato MSI in quanto deve attendere l'effettivo rimpiazzamento “forzato” da parte della L1-X.

La L1-X, si comporta come descritto in §4.2.3: all'arrivo della GETX invia verso la L2-X il messaggio EJECT, se il blocco di cache si trova nello stato E; oppure invia verso la L2-X la copia del blocco di memoria modificato, incapsulata nel messaggio PUTX, se il blocco di cache si trova nello stato M; andando poi a transire nello stato MI.

La L2-X, nel caso in cui dovesse ricevere la PUTX contenente il blocco di memoria modificato, lo memorizza; ed in ogni caso, anche quando avesse ricevuto il messaggio EJECT, lo invia attraverso una PUTX, contenente anche l'identificatore del banco di L2-X che sta rimpiazzando, alla DirectoryCache; imposta il flag L1Invalid a true; invia il messaggio WB_ACK alla L1-X; invia la copia del blocco di memoria, incapsulato nel messaggio DATA_ECLUSIVE, al banco di L2-Y richiedente il cui identificatore è memorizzato nel campo Exclusive; ed infine transisce nello stato MI.

La Dir$, ricevuto PUTX dalla L2-X, risponde al mittente con il WB_ACK sulla *virtual nertwork* ordinata; senza transire di stato ed ignorando l'eventuale blocco di memoria ricevuto poichè ormai la L2-X non è più proprieratia del blocco.

La L2-X, all'arrivo del WB_ACK proveniente dalla Dir$, conclude il rimpiazzamento, facendo transire il blocco di cache nello stato I. La L1-X allo stesso modo, all'arrivo del WB_ACK proveniente dalla L2-X, conclude il rimpiazzamento facendo transire il blocco di cache nello stato I. La L2-Y, alla ricezione del DATA_EXCLUSIVE, conclude il protocollo come descritto in §4.4.2.

Capitolo 5

Simulazioni e risultati

In questo capitolo saranno presentati i risultati ottenuti simulando un sistema CMP basato sul paradigma PS-NUCA descritto nei capitoli precedenti. In particolare il paragrafo 5.1 illustra la metodologia di simulazione e analisi adottata. Il paragrafo 5.2 si occupa di illustrare e discutere i risultati confrontandoli con analoghe simulazioni effettuate su altri sistemi CMP basati su paradigmi S-NUCA, D-NUCA e Re-NUCA. Quest'ultimo è un sistema basato sul paradigma D-NUCA in cui è stato aggiunto il meccanismo della replicazione per evitare il fenomeno del ping-pong.

5.1 Metodologia di simulazione e analisi

Nel seguente paragrafo saranno descritti gli strumenti adottati per realizzare la simulazione, le caratteristiche del sistema complessivo oltre che dei benchmark adottati.

5.1.1 Lo strumento: Simics

Simics [24] è una piattaforma di simulazione di tipo full-system (cioè, è in grado di simulare l'esecuzione di una macchina reale, su cui è possibile fare il boot di un sistema operativo, e quindi eseguire comandi e lanciare applicativi) in cui viene simulata l'esecuzione di un determinato set di istruzioni hardware; allo stato attuale, Simics è in grado di simulare l'esecuzione di differenti set di istruzioni, quali ed esempio la famiglia x86 e lo SPARC. Simics non simula né l'esecuzione out-of-order delle istruzioni, né il comportamento delle gerarchie di memoria. Per questi scopi, è fornita un'interfaccia che rende possibile il caricamento di moduli esterni che svolgano queste funzionalità. La suite GEMS [25] è costituita da due moduli che sono collegabili a Simics tramite l'interfaccia sopra citata: Opal (per la simulazione di processori out-of-order) e Ruby (per la simualzione delle gerarchie di memoria). In questo lavoro di tesi, è stato trattato solo il sottosistema di gerarchie di

memoria: pertanto, l'unico modulo che è stato considerato è Ruby.

Un altro tool utilizzato per determinare i parametri del sistema è CACTI 5.1 [26]: si tratta di un tool di memory compiling che prende in ingresso le caratteristiche costruttive della memoria e restituisce come uscita varie caratteristiche di comportamento della memoria, in particolare i suoi tempi di risposta. In questo lavoro di tesi, è stato utilizzato lo strumento CACTI per ricavare le latenze e le dimensioni fisiche (altezza e larghezza) dei banchi di cache presenti nel sistema simulato. Inoltre, mediante un semplice modello di ritardo sui fili, è stato calcolato il ritardo di propagazione sui link della NoC, utilizzando come lunghezza degli stessi le dimensioni dei banchi della NUCA fornite da CACTI.

5.1.2 Caratteristiche del sistema

Il sistema adottato è un CMP costituito da 8 processori UltraSPARC II con frequenza di clock di 5 Ghz, ognuno dei quali possiede una cache L1 privata, da una cache di secondo livello orgnizzata secondo il paradigma PS-NUCA e da una Directory Cache. Ciascuna L1 ha dimensione di 32 Kbytes, di cui 16 Kbytes di cache istruzioni e 16 Kbytes di cache dati, associative a due vie, le cui n latenze di accesso sono pari ad 1 ciclo di clock. La cache di secondo livello è una PS-NUCA di 16 Mbytes costituita da una matrice 8x8 di banchi. Ciascun banco della PS-NUCA ha dimensione pari a 256 Kbytes ed è associativo a 4 vie. Le latenze di accesso sono in tal caso di 5 cicli di clock per il campo TAG e 8 cicli per il campo TAG+DATA (latenze calcolate con CACTI). La PS-NUCA è quindi organizzata in 8 cache private, ciascuna associata ad un processore, da 8 banchi ciascuno. La cache per la Directory è di 896 Kbytes ed è composta da 8 banchi. Ciascun banco della Directory Cache ha dimensione pari a 112 Kbytes ed è associativo a 2 vie. La latenza di accesso del campo TAG è di 5 cicli di clock. Gli switch di interconnessione tra i vari link di comunicazione hanno latenza pari a 1 ciclo di clock. La Directory invece si trova in memoria principale (Off-Chip) che ha una capienza di 2 Gbytes ed è composta anch'essa da 8 banchi. Ciascun campo di memoria ha dimensione pari a 256 Mbytes. La latenza di accesso è di 300 cicli di clock.

Sulla base della dispozione dei processori è stata individuata la configurazione del sistema per le simulazioni: **4+4p**, in cui i processori sono disposti a gruppi di 4 sui lati opposti della PS-NUCA.

La Tabella 1 riassume i parametri di configurazione del sistema considerato.

5.1.3 Condizione di Esecuzione

Tabella 1

CPU	8 cpu (Ultra Sparc II), in-order
Frequenza di Clock	~5 GHz
L1 Cache	Privata 16 Kbytes Istruzioni + 16 Kbytes Dati 2-Way Set Associative Latenza di Accesso: 1 ciclo di clock
L2 Cache	16 Mbytes PS-NUCA 64 banchi da 256 Kbytes (8 privati a processore) 4 -Way Set Associative Latenza di Accesso TAG: 5 cicli di clock Latenza di Accesso TAG+DATA: 8 cicli di clock
Dimensione Blocco di Memoria	64 bytes
Directory Cache	896 Kbytes 8 banchi da 112 Kbytes 2-Way Set Associative Latenza di Accesso TAG: 5 cicli di clock
Dimensione Blocco di Directory	28 bits
Configurazione NoC	Partial 2D Mesh Network Latenza Switch: 1 ciclo di clock Latenza Link: 1 ciclo di clock
Main Memory	2 Gbytes 8 banchi da 256 Mbytes Latenza di Accesso: 300 cicli di clock

Parametri di Configurazione

Nella valutazione del sistema, sono state effettuate simulazioni di tipo full-system dove il sistema operativo è Solaris 10 della Sun Microsystems. Come Benchmarks sono state utilizzate 10 applicazioni multithreaded appartenenti a 2 famiglie:

- SPLASH-2 [28]: Ocean, Barnes, Radix, Radiosity e Raytrace;
- Parsec v2.0 [18]: Blackscholes, Bodytrack, Canneal, Streamcluster e Swaptions;

Come porzione di esecuzione è stato considerato un run di 800M di istruzioni eseguite, preceduti da una fase di warmup di 5M di istruzioni (serve per precaricare buona parte dei blocchi di memoria che servono alle applicazioni nella cache). La fase di warmup inizia subito dopo l'avvenuta parallelizzazione dell'applicazione, cioè dopo che sono stati creati i thread (numericamente pari al numero di processori del sistema). È opportuno sottolineare che i thread dell'applicazione non migrano, cioè vengono assegnati staticamente ciascuno a un processore differente.

5.2 Risultati

In questo paragrafo saranno descritti i risultati ottenuti dalle simulazioni per i benchamrk applicativi elencati in §5.1.3 ed il confronto verrà effettuato relativamente alle prestazioni.

5.2.1 Confronto sulle prestazioni

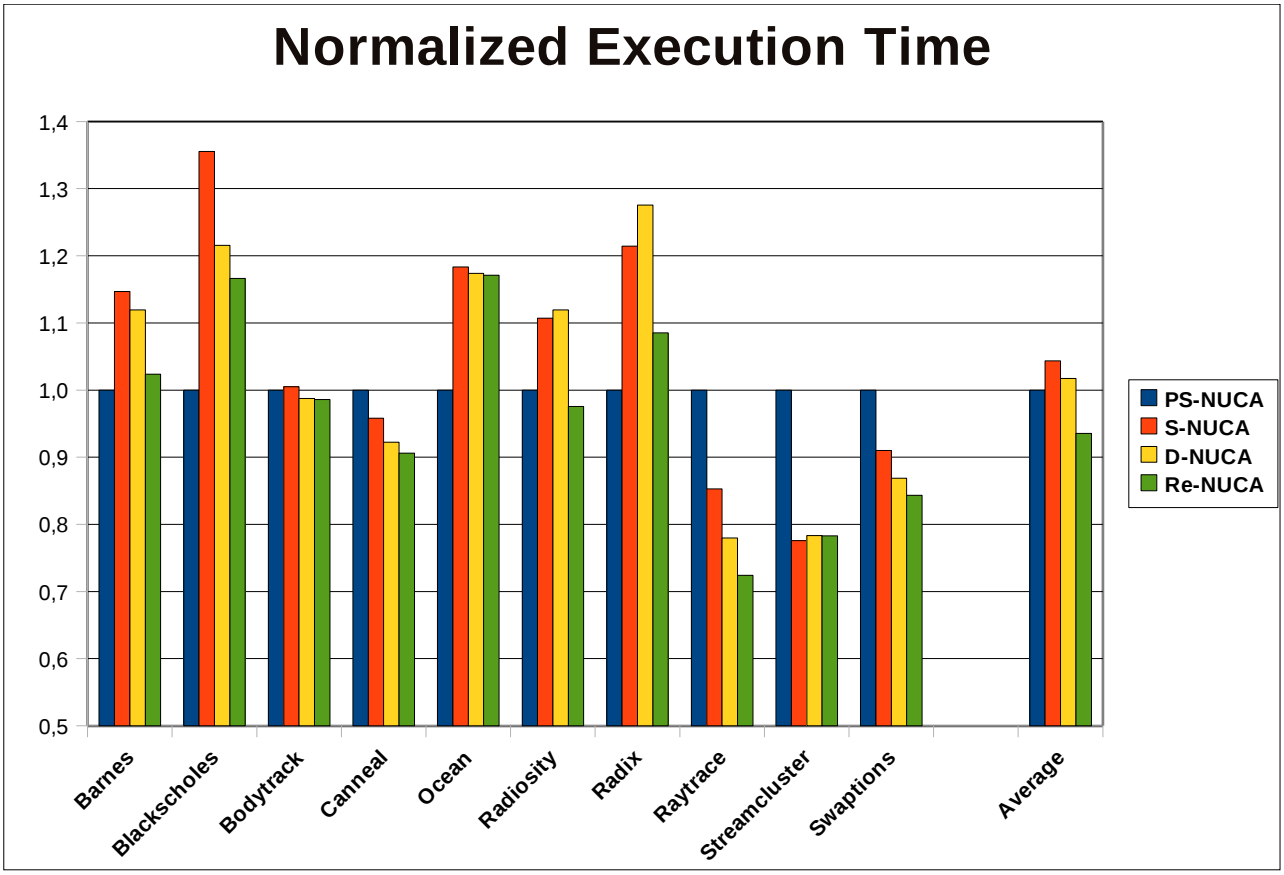

Figura 39 Normalized Execution Time

La Figura 39 illustra il Normalized Execution Time per tutti i benchmarks considerati, il cui il sistema PS-NUCA è stato preso come riferimento. Dall'analisi dei risultati riportati in figura si nota

che mediamente le prestazioni sono migliori rispetto ai paradigmi S-NUCA e D-NUCA, rispettivamente di circa il 4% e 2%, mentre peggiorano rispetto alla Re-NUCA di circa il 7%. Dalla figura si osserva anche che vi sono benchmarks in cui la PS-NUCA offre prestazioni migliori (Barnes, Blackscholes, Ocean e Radix) ed altri in cui c'è un netto peggioramento (Canneal, Raytrace, Streamcluster e Swaptios) rispetto agli altri paradigmi. Le prestazioni rispecchiano un bilanciamento sul Miss Rate in L1 Cache (Figura 40), sul Miss Rate in L2 Cache (Figura 41), e sulla Latenza Media di Risposta in prensenza di Miss in L1 Cache (Figura 42).

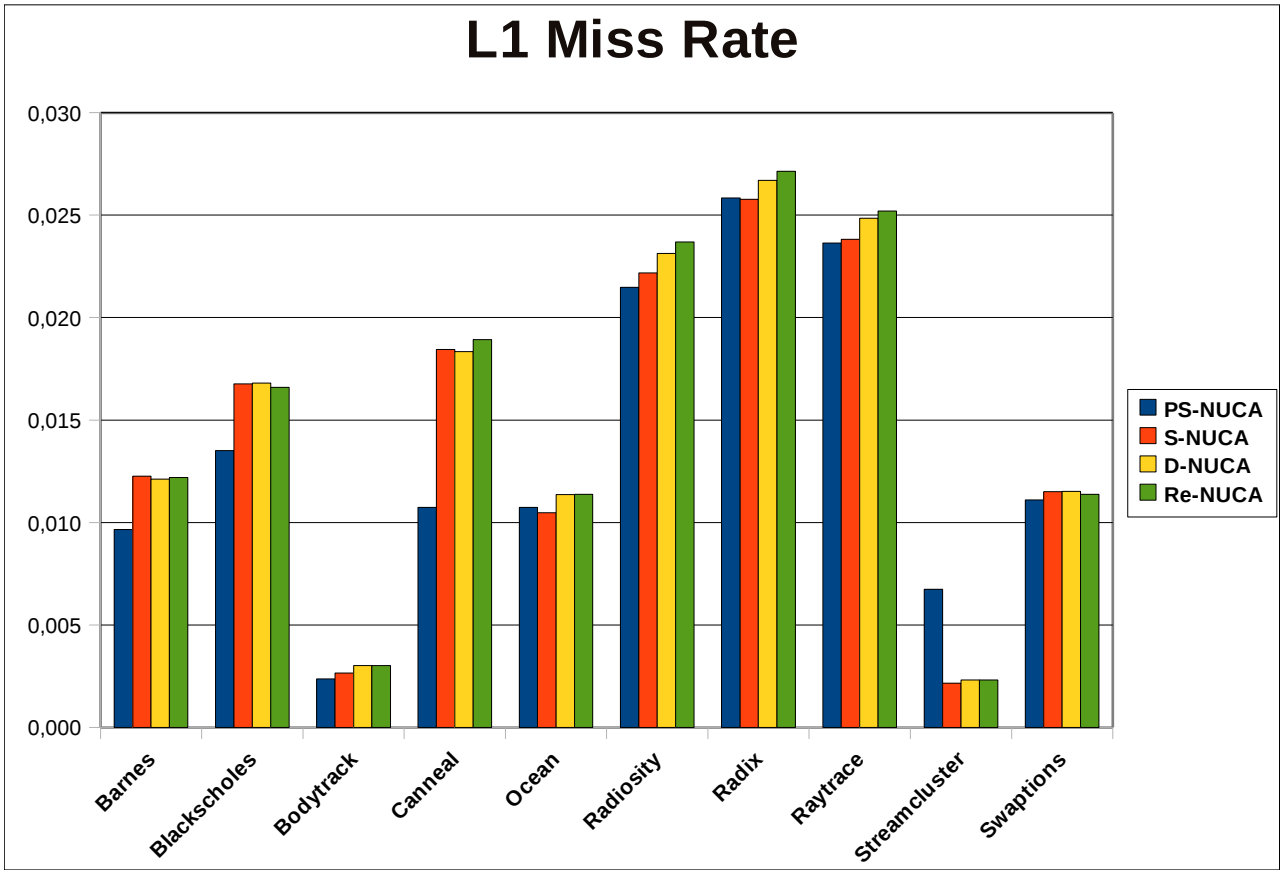

Figura 40 L1 Miss Rate

In particolare quest'ultimo parametro, come illustrato in Figura 43, è una media pesata sugli accessi di 3 Latenze Medie di Risposta:

- Latenza Media di Risposta in presenza di Miss in L1 Cache ed Hit in L2 Cache;
- Latenza Media di Risposta in presenza di Miss in L1 Cache con trasferimento On-Chip del blocco di memoria: ossia la copia del blocco viene fornita direttamente da un altro banco di cache che la possiede;

- Latenza Media di Risposta in presenza di Miss in L1 Cache e Miss in L2 Cache, con trasferimento Off-Chip del blocco di memoria: ossia la copia del blocco va recuperata direttamente in memoria principale.

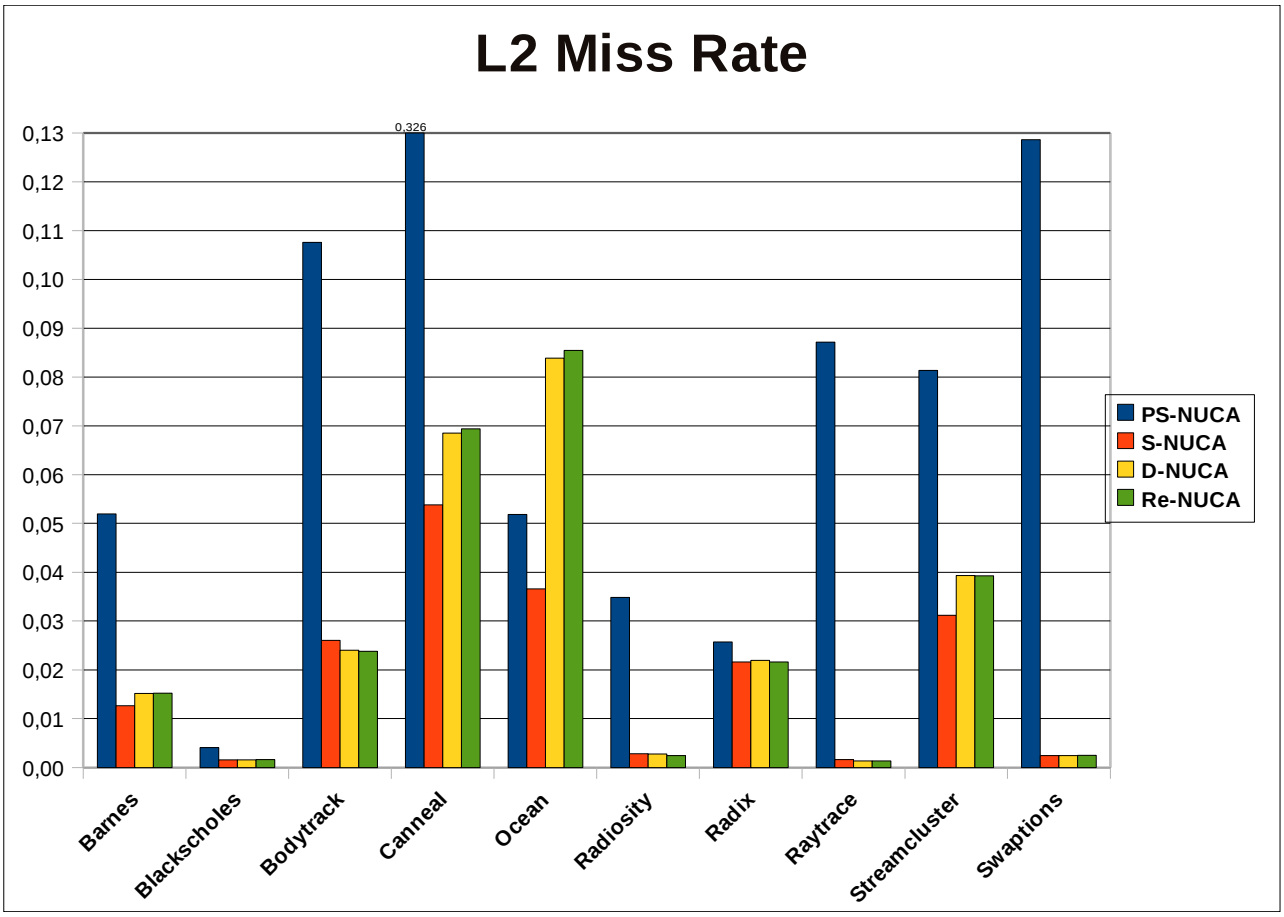

Figura 41 L2 Miss Rate

Per quanto riguarda Barnes, il miglioramento delle prestazioni è dovuto al fatto che, anche se si è in presenza di un elevato Miss Rate in L2$ (circa il 400% di miss in più rispetto agli altri paradigmi), si ha una significativa riduzione del Miss Rate in L1$ (circa il 20%) e da una più bassa Latenza Media di Risposta in presenza di Miss in L1 dovuta la fatto che si riscontrano più Hit in L2$.

Per Blackscholes il discorso è differente: il miglioramento delle prestazioni è dovuto dalla significativa riduzione del Miss Rate in L1 (circa il 25%) in quanto il Miss Rate in L2$, essendo inferiore all'1%, non incide sulle prestazioni.

Bodytrack ha un Miss Rate in L1 Cache molto ridotto e comparabile con tutti gli altri paradigmi. Questo comporta che le prestazioni siano pressochè identiche.

Su Canneal la PS-NUCA perde in prestazioni rispetto agli altri paradigmi anche se si è in presenza di una elevata riduzione del Miss Rate in L1$ (circa il 70%). Ciò è dovuto all'elavato Miss Rate in L2$ (30% in più rispetto agli altri paradigmi) e all'elevata Latenza Media di Risposta in presenza di Miss in L1$ dovuta al fatto che, nella maggior parte delle volte, bisogna andare a recuperare il blocco in memoria principale.

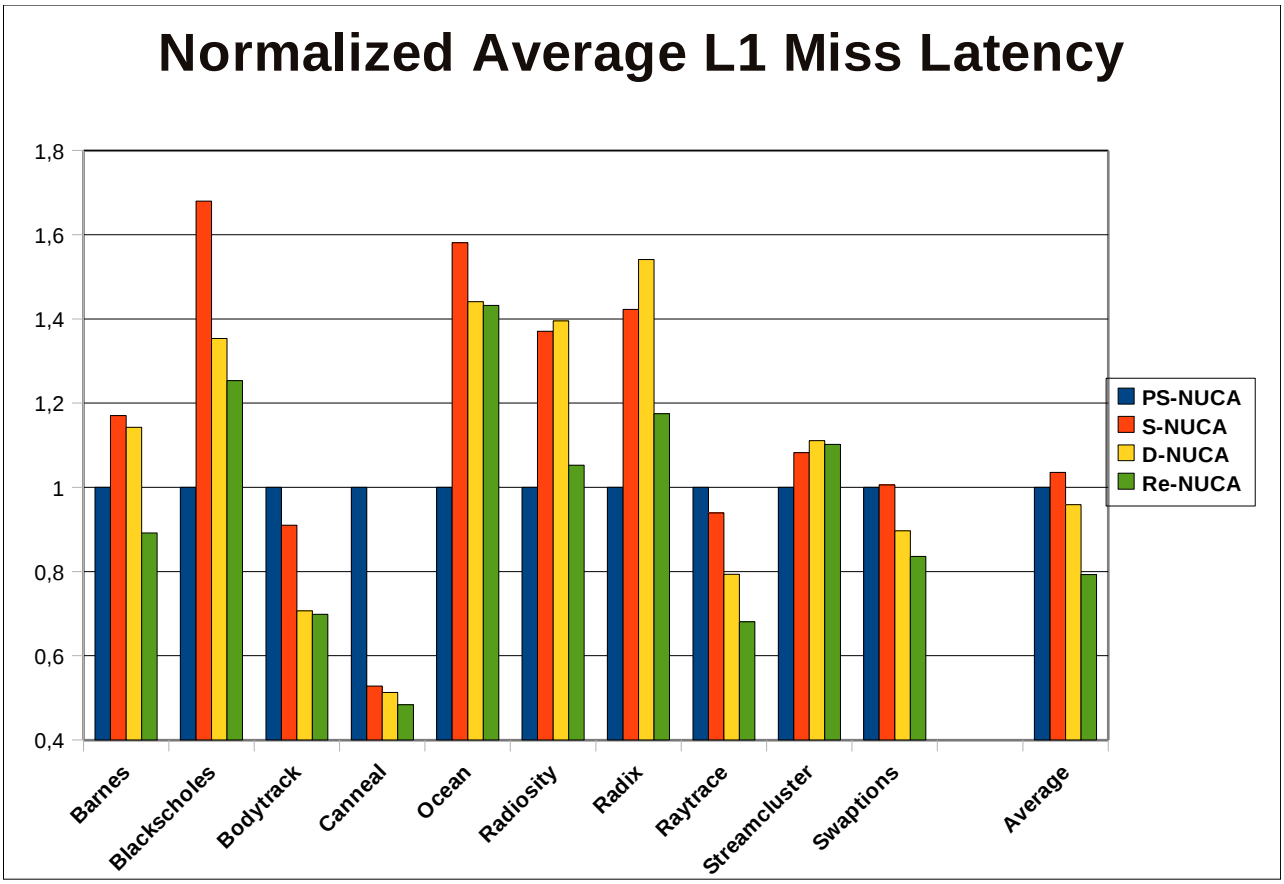

Figura 42 Normalized Average L1 Miss Latency

Per quanto rigurda il Benchmark Ocean, si osserva che un netto miglioramento delle prestazioni della PS-NUCA rispetto gli altri 3 paradigmi. Questo perchè tale Benchmark, anche se ha uno spazio di memoria condiviso, vi accede, per mezzo dei propri threads, in modo privato: ossia ogni thread accede ad una propria fetta di dati. I paradigmi D-NUCA e Re-NUCA soffrono questo comportamento e ne risentono in termini di Miss Rate in L2$ (cica il 60% più elevato rispetto alla PS-NUCA). Difatti, per tali paradigmi, ogni blocco di memoria, proveniente dalla memoria

principale, arriva in L2$ in un unico banco di cache, per ognuno degli 8 *bankset*, chiamato collettore. Se tale blocco non viene effettivamente condiviso tra i vari threads, non riesce a migrare verso i banchi più vicini ai processori; per cui, con molta probabilità, verrà rimpiazzato [31]. Questo comportamento, invece, favorisce il paradigma S-NUCA in cui il blocco di memoria può essere memorizzato in ognuno dei 64 banchi di L2$, mentre nella PS-NUCA solo negli 8 considerati privati. Quest'ultima però guadagna in prestazioni sulla semplice S-NUCA a causa della più bassa Latenza Media di Rsposta in presenza di Miss in L1 Cache (circa il 60% in meno).

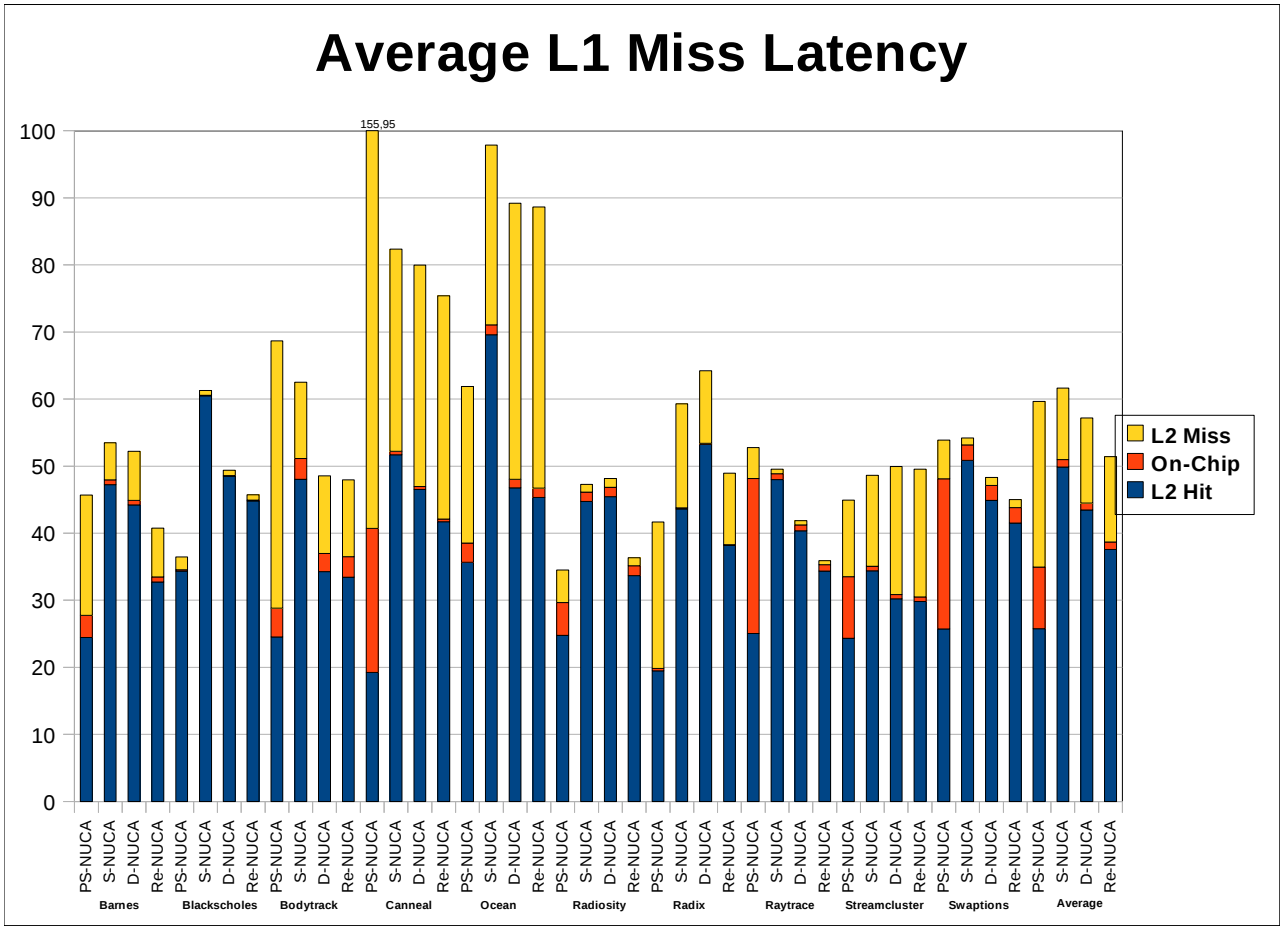

Figura 43 Average L1 Miss Latency

Radiosity è l'unico Benchmark che rispecchia l'andamento medio delle prestazioni sui 4 paradigmi. La PS-NUCA guadagna sulla S-NUCA, D-NUCA e Re-NUCA in termini di Miss Rate in L1$ mentre perde sul Miss Rate in L2$ (circa il 30%) e questo provoca l'abbattimento delle prestazioni. Nonostante tutto però, guadagna sui paradigmi S-NUCA e D-NUCA, in quanto la Latenza Media di Risposta in presenza di Miss in L1 Cache è significativamente inferiore (di circa il 50%).

Radix presenta un Miss Rate in L1$ ed in L2$ abbastanza elevato (circa il 25%) e comparabile tra tutti e 4 i paradigmi. Il miglioramento delle prestazione per la PS-NUCA è dovuta alla più bassa Latenza Media di Risposta in presenza di Miss in L1$.

Raytrace è il Benchmark in cui la PS-NUCA ha prestazioni inferiori rispetto alla S-NUCA, D-NUCA e Re-NUCA. Il Miss Rate in L1$ è elevato e comparabile in tutti i paradigmi (circa 25%), ma è il Miss Rate in L2$ che provoca il decadimento delle prestazioni: esso risulta essere circa l'800% superiore agli altri paradigmi. Quest'ultimo effetto è leggermente attenuato dalla Latenza Media di Risposta in presenza di Miss in L1$.

Streamcluster è comparabile in prestazione al Benchmark Raytrace. Le prestazioni della PS-NUCA sono leggermente migliori in quanto il Miss Rate in L2$ risulta essere "solo" il doppio rispetto a quello degli altri paradigmi. Inoltre la Latenza Media di Risposta in presenza di Miss in L1$ risuta essere circa il 10% inferiore.

In fine Swaptions: il Miss Rate in L1$ è comparabile tra tutti e 4 i paradigmi (circa 12%); Le prestazioni decadono a causa dell'elevatissimo Miss Rate in L2$.

Capitolo 6

Conclusioni e sviluppi futuri

L’elevato livello di integrazione raggiunto nelle moderne tecnologie a semiconduttore, consente di aumentare il numero di transistor in un chip di silicio. Questo ha permesso la realizzazione di sistemi multiprocessore su un singolo chip (multicore o CMP), nei quali i processori condividono una gerarchie di memorie cache di basso livello e di grosse dimensioni. Poiché le prestazioni di una cache convenzionale sono limitate dagli effetti del *wire delay*, la necessità di comunicazione a bassa latenza e larga banda nei multicore, ha portato allo studio di nuove architetture per la cache condivisa. In particolare le cache di tipo NUCA si sono dimostrate efficaci al riguardo.

In questo lavoro di tesi è stata progettata, implementata e simulata una gerarchia di cache di tipo PS-NUCA in un sistema multicore. Una volta implementata e testata l’architettura si è passati alla simulazione della stessa attraverso l’uso di Simics e Ruby in riferimento a 10 applicazioni facenti parte delle suite SPLASH-2 e Parsec- v2.0. I risultati ottenuti dalle simulazioni sono stati analizzati per verificare, da un lato, il corretto funzionamento della PS-NUCA e del protocollo di coerenza, dall’altro, per valutare le prestazioni del nuovo sistema.

In particolare, dall’analisi dei dati sulle performances si è notato che il sistema PS-NUCA in media guadagna in prestazioni rispetto alla S-NUCA ed alla D-NUCA, mentre perde in media rispetto alla Re-NUCA. Questo perchè, essendo la L2 Cache ripartita in 8 cache private ad ogni processore, comporta un aumento del Miss Rate in L2 Cache (Miss di Capacità). Si perde, rispetto alla D-NUCA ed alla Re-NUCA ma non rispetto alla semplice S-NUCA, anche in termini di Latenza Media di Risposta in presenza di Miss in L1 Cache poichè, a differenza degli altri 2 paradigmi, la PS-NUCA è caratterizzata dal fatto che la Directory giace in memoria principale e che si necessita di una Directory Cache di dimensioni elevate per evitare l'accesso Off-Chip. Inoltre per come è strutturato il protocollo di coerenza MESI, la PS-NUCA è svantaggiata sul recupero di blocchi condivisi poichè, quando un processore richiede un tale blocco di memoria, presente in altri banchi di L2 Cache privati di processori, il protocollo comporta un accesso in memoria principale.

Tale effetto è possibile ridurlo o addirittura eliminarlo progettando sulla PS-NUCA un protocollo di tipo MOESI caratterizzato dal fatto che si elimina la ownership della Directory per ogni blocco di memoria la cui copia è memorizzata in almeno un banco di L2 Cache.

Appendice A

Stati del protocollo MESI PS-NUCA

A.1 L1 Cache

- **M**: **Modified**. Il blocco di memoria, memorizzato nella L1Cache, è stato modificato, per mezzo di operazioni di Store da parte del processore, rispetto alla copia memorizzata nel relativo banco della L2Cache. Tale banco è l'unico della L2Cache ad avere memorizzato la copia del blocco.

- **E**: **Exclusive**. Il blocco di memoria, memorizzato nella L1Cache, è coerente con la copia memorizzata nel relativo banco della L2Cache. Tale banco è l'unico della L2Cache ad avere memorizzata la copia del blocco.

- **S**: **Shared**. Il blocco di memoria, memorizzato nella L1Cache, è coerente con la copia memorizzata nel relativo banco della L2Cache. Tale banco può non essere l'unico della L2Cache ad avere memorizzata la copia del blocco.

- **I**: **Invalid**. Il blocco di cache non è considerato valido.

- **MS**: **Modified to Shared**. Il blocco di memoria, memorizzato nella L1Cache, si trovava nello stato M oppure E. A seguito della ricezione del messaggio `GETS` oppure `GET_INSTR` proveniente dalla L2Cache, ha risposto a quest'ultima con un messaggio `PUTS` (contenete la copia del blocco) oppure `ACCEPTS` ed attende, sempre dalla L2Cache, un messaggio `WB_ACK` per poter transire nello stato S.

- **MI**: **Modified to Invalid**. Il blocco di memoria, memorizzato nella L1Cache, si trovava nello stato M, E oppure MS. La L1Cache è costretta ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L1Cache stessa, oppure a causa della ricezione di un messaggio `GETX` dalla L2Cache. Ha già inviato un messaggio `PUTX` (contenente la copia del blocco) oppure un messaggio `EJECT` verso la L2Cache; ed attende, da quest'ultima, un messaggio `WB_ACK` per poter transire nello stato I.

- **IM**: **Invalid to Modified**. Il blocco di memoria, se memorizzato nella L1Cache, si trovava nello stato S, altrimenti la L1Cache è nello stato I. Avendo ricevuto richiesta da parte del Processore per poter effettuare operazioni di Store, ha inviato un messaggio `GETX` verso la L2Cache ed attende di ricevere da quest'ultima la copia del blocco di memoria in modo esclusivo per poter transire nello stato M.

- **IE**: **Invalid to Exclusive**. La L1Cache si trovava nello stato I. Avendo ricevuto richiesta da parte del Processore per poter effettuare operazioni di Load oppure di iFetch, ha inviato un messaggio `GETS` oppure `GET_INSTR` verso la L2Cache ed attende di ricevere da quest'ultima la copia del blocco di memoria in modo condiviso oppure in modo esclusivo per poi poter transire nello stato S oppure nello stato E.

- **IMS**: **Invalid to Modified to Shared**. La L1Cache si trovava nello stato IM ed ha ricevuto un messaggio `GETS` oppure `GET_INTSR` da parte della L2Cache. Attende il blocco di memoria in modo esclusivo proveniente dalla L2$ per permettere al Processore di effettuare l'operazione di Store richiesta. Dovrà poi reinviare, per mezzo del messaggio `PUTS`, il blocco modificato alla L2$ stessa, che le invierà un `WB_ACK`, per poter transire nello stato S.

- **IMI**: **Invalid to Modified to Invalid**. La L1Cache si trovava nello stato IM oppure IMS ed è costretta ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L1Cache stessa, oppure a causa della ricezione di un messaggio `GETX` dalla L2Cache. Attende il blocco di memoria in modo esclusivo proveniente dalla L2$ per permettere al Processore di effettuare l'operazione di Store richiesta. Dovrà poi reinviare, per mezzo del messaggio `PUTX`, il blocco modificato alla L2$ stessa, che le invierà un `WB_ACK`, per poter transire nello stato I.

- **IES**: **Invalid to Exclusive to Shared**. La L1Cache si trovava nello stato IE ed ha ricevuto un messaggio `GETS` oppure `GET_INTSR` da parte della L2Cache. Attende il blocco di memoria in modo esclusivo proveniente dalla L2$ per permettere al Processore di effettuare l'operazione di Load oppure di iFetch richiesta. Dovrà poi inviare un messaggio `ACCEPTS` alla L2$ stessa, che le invierà un `WB_ACK`, per poter transire nello stato S.

- **IEI**: **Invalid to Exclusive to Invalid**. La L1Cache si trovava nello stato IE oppure IES ed è costretta ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L1Cache stessa, oppure a causa della ricezione di un messaggio `GETX` dalla L2Cache. Attende il blocco di memoria in modo esclusivo o condiviso proveniente dalla L2$ per permettere al Processore di effettuare l'operazione di Load oppure di iFetch richiesta. Dovrà poi inviare un messaggio `EJECT` alla L2$ stessa, che le invierà un `WB_ACK`, per poter transire nello stato I.

A.2 L2 Cache

- **M**: **Modified**. Il blocco di memoria, memorizzato nel banco della L2Cache, è stato modificato, per mezzo di un rimpiazzamento (`PUTX`) della L1Cache, rispetto alla copia che risiede in memoria principale. Tale banco è l'unico della L2Cache ad avere memorizzata la copia del blocco.

- **E**: **Exclusive**. Il blocco di memoria, memorizzato nel banco della L1Cache, è coerente con la copia che risiede in memoria principale. Tale banco è l'unico della L2Cache ad avere memorizzata la copia del blocco.

- **S**: **Shared**. Il blocco di memoria, memorizzato nel banco della L2Cache, è coerente con la copia che risiede in memoria principale. Tale banco può non essere l'unico della L2Cache ad avere memorizzata la copia del blocco.

- **I**: **Invalid**. Il blocco di cache non è considerato valido.

- **MS**: **Modified to Shared**. Il blocco di memoria, memorizzato nel banco della L2Cache, si trovava nello stato M oppure E ed ha il flag `L1Invalid` impostato a `true`. Ha ricevuto un messaggio `GETS` oppure `GET_INSTR`, proveniente dalla DirectoryCache, contenente l'identificatore del banco della L2$ richiedente la copia condivisa del blocco di memoria. Ha risposto alla Dir$ con un messaggio `PUTS`, contenente la copia del blocco, oppure `ACCEPTS`. Ha inviato la copia del blocco di memoria, per mezzo di un messaggio `DATA_SHARED` al banco di L2$ richiedente e attende, sempre dalla Dir$, un messaggio `WB_ACK` per poter transire nello stato S. E' in grado di servire tutte le richieste di tipo `GETS` oppure `GET_INSTR` provenienti sia dalla L1$, sia dalla Dir$.

- **MI: Modified to Invalid**. Il blocco di memoria, memorizzato nel banco della L2Cache, si trovava indifferentemente nello stato M, E, oppure MS con il flag `L1Invalid` impostato a `true`. Il banco della L2$ è costretto ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L2$ stessa, oppure a causa della ricezione di un messaggio `GETX` dalla DirectoryCache. Ha già inviato un messaggio `PUTX` oppure PUTS (contenente la copia del blocco), oppure un messaggio `EJECT` oppure `ACCEPTS` verso la Dir$; ed attende, da quest'ultima, un messaggio `WB_ACK` per poter transire nello stato I. E' in grado di servire tutte le richieste di tipo `GETS` oppure `GET_INSTR` provenienti solo dalla Dir$ e non dalla L1$.

- **IM**: **Invalid to Modified**. Il blocco di memoria, se memorizzato nel banco della L2Cache, si trovava nello stato S, altrimenti è nello stato I. Avendo ricevuto un messaggio `GETX` da parte della L1$, ha inviato un messaggio `GETX` verso la DirectoryCache. Attende di ricevere la copia del blocco in modo esclusivo, oppure in modo condiviso insieme agli acknowledgement di avvenuta invalidazione da parte degli altri banchi della L2$ che condividevano la copia, per poter inviare la copia stessa alla L1$, ed infine transire nello stato M.

- **IE**: **Invalid to Exclusive**. Il banco della L2Cache si trovava nello stato I. Avendo ricevuto un messaggio `GETS` oppure `GET_INSTR` proveniente dalla L1Cache, ha inviato un messaggio `GETS` oppure `GET_INSTR` verso la DirectoryCache ed attende di ricevere la copia del blocco di memoria in modo condiviso oppure in modo esclusivo per poi poter transire nello stato S oppure nello stato E.

- **MSS: Modified to Shared waiting for L1-Cache Sharing**. Il blocco di memoria, memorizzato nel banco della L2Cache, si trovava nello stato M oppure E ed ha il flag `L1Invalid` impostato a `false`. Ha ricevuto un messaggio `GETS` oppure `GET_INSTR`, proveniente dalla DirectoryCache, contenente l'identificatore del banco della L2$ richiedente la copia condivisa del blocco di memoria. Ha memorizzato il suddetto identificatore. Ha inoltrato il messaggio ricevuto alla L1Cache e attende da quest'ultima la copia del blocco per poterla inviare ai banchi di L2$ richiedenti ed alla Dir$ per poi impostare il flag `L1Invalid` a `true` e transire in MS.

- **MSI: Modified to Shared waiting for L1-Cache Invalidation**. Il blocco di memoria, memorizzato nel banco della L2Cache, si trovava indifferentemente nello stato M, E, oppure MSS con il flag `L1Invalid` impostato a `false`. Il banco della L2$ è costretto ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L2$ stessa, oppure a causa della ricezione di un messaggio `GETX` dalla DirectoryCache che ha prontamente inoltrato alla L1Cache dopo aver memorizzato l'identificatore del banco della L2$ richiedente. Attende dalla L1$ la copia del blocco per poterla inviare ai banchi di L2$ richiedenti ed alla Dir$ per poi impostare il flag `L1Invalid` a `true` e transire in MI.

- **IMS**: **Invalid to Modified to Shared**. Il banco della L2Cache si trovava nello stato IM ed ha ricevuto un messaggio `GETS` oppure `GET_INTSR` da parte della DirectoryCache. Ha memorizzato l'identificatore del banco della L2$ richiedente. Ha inoltrato il messaggio ricevuto alla L1Cache. Attende di ricevere la copia del blocco in modo esclusivo, oppure in modo condiviso insieme agli acknowledgement di avvenuta invalidazione da parte degli altri banchi della L2$ che condividevano la copia, per poterla inviare alla L1$. Dovrà attendere poi che la L1$ rimandi indietro la copia modificata per poterla inoltrare ai banchi di L2$ richiedenti ed alla Dir$, che le risponderà con un `WB_ACK`, per poter poi transire nello stato S.

- **IMI**: **Invalid to Modified to Invalid**. Il banco della L2Cache si trovava nello stato IM oppure IMS ed è costretta ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L2$ stessa, oppure a causa della ricezione di un messaggio `GETX` dalla DirectoryCache che ha prontamente inoltrato alla L1Cache dopo aver memorizzato l'identificatore del banco della L2$ richiedente. Attende di ricevere la copia del blocco in modo esclusivo, oppure in modo condiviso insieme agli acknowledgement di avvenuta invalidazione da parte degli altri banchi della L2$ che condividevano la copia, per poterla inviare alla L1$. Dovrà attendere poi che la L1$ rimandi indietro la copia modificata per poterla inoltrare ai banchi di L2$ richiedenti ed alla Dir$, che le risponderà con un `WB_ACK`, per poter poi transire nello stato I.

- **IES**: **Invalid to Exclusive to Shared**. Il banco della L2Cache si trovava nello stato IE ed ha ricevuto un messaggio `GETS` oppure `GET_INTSR` da parte della DirectoryCache. Ha memorizzato l'identificatore del banco della L2$ richiedente. Ha inoltrato il messaggio ricevuto alla L1Cache. Attende la copia del blocco di memoria in modo esclusivo proveniente dalla memoria principale per poterla inviare alla L1$. Invierà poi `ACCEPTS` alla Dir$. Attenderà che la L1$ le invii un messaggio `ACCEPTS` oppure `EJECT` per poter inviare la copia anche ai banchi di L2$ richiedenti, dopo che la Dir$ le avrà risposto con `WB_ACK`, eppoi infine transire nello stato S.

- **IEI**: **Invalid to Exclusive to Invalid**. Il banco della L2Cache si trovava nello stato IE oppure IES ed è costretta ad effettuare un rimpiazzamento del blocco, che potrebbe essere stato imposto dall'algoritmo LRU implementato nella L2$ stessa, oppure a causa della ricezione di un messaggio `GETX` dalla DirectoryCache che ha prontamente inoltrato alla L1Cache dopo aver memorizzato l'identificatore del banco della L2$ richiedente. Attende la copia del blocco di memoria in modo esclusivo oppure condivido per poterla inviare alla L1$. Invierà poi `ACCEPTS` alla Dir$. Attenderà che la L1$ le invii un messaggio `ACCEPTS` oppure `EJECT` per poter inviare la copia anche ai banchi di L2$ richiedenti, dopo che la Dir$ le avrà risposto con `WB_ACK`, eppoi infine transire nello stato I.

A.3 Directory Cache

- **P**: **Private**. La copia del blocco di directory è presente nella DirectoryCache e la copia del relativo blocco di memoria è memorizzato, nello stato E oppure M, in un solo banco della L2Cache.

- **S**: **Shared**. La copia del blocco di directory è presente nella DirectoryCache e la copia del relativo blocco di memoria è stata inviata in modo condiviso ai banchi della L2Cache i cui identificatori sono memorizzati nella lista degli sharers.

- **I**: **Invalid**. La copia del bocco di directory è presente nella DirectoryCache e la copia del relativo blocco di memoria non è presente in nessun banco della L2Cache.

- **PS**: **Private to Shared**. La copia del blocco di directory è presente nella DirectoryCache e la copia del relativo blocco di memoria è memorizzato, nello stato E oppure M, in un solo banco della L2Cache. Uno o più ulteriori banchi della L2$ ha fatto richiesta in modo condiviso per il blocco di memoria.

- **R**: **Replaced**. La copia del blocco di directory non è presente nella DirectoryCache.

- **F**: **Fetch**. La copia del blocco di directory non è presente nella DirectoryCache. Quest'ultima ha inviato una richiesta per una copia del blocco di directory alla Directory in memoria principale poiché vi sono richieste pendenti, provenienti da banchi della L2Cache, per il relativo blocco di memoria.

A.4 Directory

- **C**: **Cached**. La copia aggiornata del blocco di directory è presente nella DirectoryCache.

- **P**: **Private**. La copia aggiornata del blocco di directory non è presente nella DirectoryCache e la copia del relativo blocco di memoria è memorizzato, nello stato E oppure M, in un solo banco della L2Cache.

- **S**: **Shared**. La copia aggiornata del blocco di directory non è presente nella DirectoryCache e la copia del relativo blocco di memoria è stata inviata in modo condiviso ai banchi della L2Cache i cui identificatori sono memorizzati nella lista degli sharers.

- **I**: **Invalid**. La copia aggiornata del bocco di directory non è presente nella DirectoryCache e la copia del relativo blocco di memoria non è presente in nessun banco della L2Cache.

- **PS**: **Private to Shared**. La aggiornata copia del blocco di directory non è presente nella DirectoryCache e la copia del relativo blocco di memoria è memorizzato, nello stato E oppure M, in un solo banco della L2Cache. Uno o più ulteriori banchi della L2$ ha fatto richiesta in modo condiviso per il blocco di memoria.

Appendice B

Eventi del protocollo MESI PS-NUCA

B.1 L1 Cache

- **Ifetch**. Il Processore necessita di effettuare l'operazione di iFetch in un determinato blocco di memoria.

- **Load**. Il Processore necessita di effettuare l'operazione di Load in un determinato blocco di memoria.

- **Store**. Il Processore necessita di effettuare l'operazione di Store in un determinato blocco di memoria.

- **Replacement**. Il blocco di cache è stato selezionato dall'algoritmo LRU per un rimpiazzamento.

- **GET_INSTR**: Get Instruction. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco istruzioni.

- **GETS**: **Get Shared**. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco dati.

- **GETX**: **Get eXclusive**. Un banco della L2Cache privata di un'altro processore fa richiesta in modo esclusivo la copia del blocco di memoria; oppure un banco della L2$ privata fa richiesta si invalidazione del blocco di memoria condiviso; oppure ancora la L2$ privata chiede un rimpiazzamento forzato del blocco di memoria esclusivo.

- **DataShared**. E' arrivata la copia di un blocco di memoria in modo condiviso.

- **Data_Exclusive**. E' arraviata la copia di un blocco di memoria in modo esclusivo.

- **WB_Ack**: **Write-Back Acknowledgement**. E' arrivata l'accettazione di rimpiazzamento per il blocco da parte della L2Cache privata.

B.2 L2 Cache

- **GET_INSTR**: **Get Instruction**. E' arrivata dalla L1Cache una richiesta per il blocco istruzioni.

- **GETS**: **Get Shared**. E' arrivata dalla L1Cache una richiesta per il blocco dati.

- **GETX**: **Get eXclusive**. E' arrivata dalla L1Cache una richiesta in modo esclusivo la copia del blocco di memoria.

- **ACCEPTS**: **Accept Shared**. E' arrivata dalla L1Cache l'accettazione per la richiesta di condivisione della copia del blocco di memoria.

- **PUTS**: **Put Shared**. E' arrivata dalla L1Cache la copia aggiornata del blocco di memoria che può essere considerato condiviso.

- **EJECT**. E' arrivata dalla L2Cache l'informazione che la copia del blocco di memoria in L1$ non è stata modificata e sta per essere invalidata.

- **PUTX**: **Put eXclusive**. E' arrivata dalla L1Cache la copia aggiornata del blocco di memoria che sta per essere invalidata.

- **Fwd_GET_INSTR**: **Forward Get Instruction**. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco istruzioni.

- **Fwd_GET_INSTR_I**: **Forward Get Instruction with L1 Invalid**. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco istruzioni e si ha la sicurezza che la copia del blocco non è presente nella L1Cache.

- **Fwd_GETS**: **Forward Get Shared**. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco dati.

- **Fwd_GETS_I**: **Forward Get Shared with L1 Invalid**. Un banco della L2Cache privata di un'altro processore fa richiesta per la condivisione del blocco dati e si ha la sicurezza che la copia del blocco non è presente nella L1Cache.

- **Fwd_GETX**: **Forward Get eXclusive**. Un banco della L2Cache privata di un'altro processore fa richiesta in modo esclusivo la copia del blocco di memoria.

- **Fwd_GETX_I**: **Forward Get eXclusive with L1 Invalid**. Un banco della L2Cache privata di un'altro processore fa richiesta in modo esclusivo la copia del blocco di memoria e si ha la sicurezza che la copia del blocco non è presente nella L1Cache.

- **Invalidate**. E' arrivata dalla DirectoryCache una richiesta di invalidazione per la copia del blocco di memoria condivisa.

- **DataShared**. E' arrivata la copia di un blocco di memoria in modo condiviso.

- **Data_Exclusive**. E' arraviata la copia di un blocco di memoria in modo esclusivo.

- **Ack**: A**cknowledgement**. E' arrivata la conferma di avvenuta invalidazione della copia del blocco di memoria condivisa memorizzata in un banco della L2Cache privata di un'altro processore.

- **LastAck**: **Last Acknowledgement**. E' arrivata la conferma di avvenuta invalidazione della copia del blocco di memoria condiviso da parte di tutti i banchi della L2Cache che la possedevano.

- **WB_Ack: Write-Back Acknowledgement**. E' arrivata l'accettazione di rimpiazzamento per il blocco da parte della DirectoryCache.

- **Replacement**. Il blocco di cache è stato selezionato dall'algoritmo LRU per un rimpiazzamento.

- **Replacement_I**: **Replacement with L1 Invalid**. Il blocco di cache è stato selezionato dall'algoritmo LRU per un rimpiazzamento e si ha la sicurezza che la copia delblocco non è presente nella L1Cache.

B.3 Directory Cache

- **GET_INSTR**: **Get Instruction**. E' arrivata da un banco della L2Cache una richiesta per il blocco istruzioni.

- **GETS**: **Get Shared**. E' arrivata da un banco della L2Cache una richiesta per il blocco dati.

- **GETX**: **Get eXclusive**. E' arrivata da un banco della L2Cache una richiesta in modo esclusivo la copia del blocco di memoria.

- **ACCEPTS_Owner**: **Accept Shared Owner**. E' arrivata, dal banco della L2Cache proprietario della copia del blocco di memoria, l'accettazione per la richiesta di condivisione della copia del blocco.

- **PUTS_Owner**: **Put Shared Owner**. E' arrivata, dal banco della L2Cache proprietario della copia del blocco di memoria, la copia aggiornata che può essere considerata condivisa.

- **EJECT_Owner**: **Eject Owner**. E' arrivata, dal banco della L2Cache proprietario della copia del blocco di memoria, l'informazione che la copia in L2$ non è stata modificata e sta per essere invalidata.

- **PUTX_Owner**: **Put eXclusive Owner**. E' arrivata, dal banco della L2Cache proprietario della copia del blocco di memoria, la copia aggiornata che sta per essere invalidata.

- **WB_NotOwner**: **Write-Back Not Owner**. E' arrivata la richiesta di rimpiazzamento per la copia del blocco di memoria da parte di un banco della L2Cache che non proprietario del blocco stesso.

- **Replacement**. Il blocco di cache è stato selezionato dall'algoritmo LRU per un rimpiazzamento.

- **DirBlock_P**: **Directory Block State P**. E' arrivato dalla Directory il blocco di directory nello stato P.

- **DirBlock_S**: **Directory Block State S**. E' arrivato dalla Directory il blocco di directory nello stato S.

- **DirBlock_I**: **Directory Block State I**. E' arrivato dalla Directory il blocco di directory nello stato I.

- **DirBlock_PS**: **Directory Block State PS**. E' arrivato dalla Directory il blocco di directory nello stato PS.

B.4 Directory

- **GETS**: **Get Shared**. E' arrivata dalla DirectoryCache una richiesta per la copia del blocco di memoria in modo condiviso.

- **GETX: Get eXclusive**. E' arrivata dalla DirectoryCache una richiesta per la copia del blocco di memoria in modo esclusivo.

- **PUTX**: **Put eXclusive**. E' arrivata dalla DirectoryCache la copia aggiornata del blocco di memoria.

- **GET**. E' arrivata dalla DirectoryCache una richiesta per la copia del blocco di directory.

- **PUT_P**: **Put Block State P**. E' arrivato dalla DirectoryCache il blocco di directory nello stato P.

- **PUT_S**: **Put Block State S**. E' arrivato dalla DirectoryCache il blocco di directory nello stato S.

- **PUT_I**: **Put Block State I**. E' arrivato dalla DirectoryCache il blocco di directory nello stato I.

- **PUT_PS**: **Put Block State PS**. E' arrivato dalla DirectoryCache il blocco di directory nello stato PS.

Appendice C

Tabelle delle transizioni di stato e relative azioni per il protocollo MESI PS-NUCA

Di seguito saranno illustrate le tabelle relative alle coppie <stato, evento> e per ciascuna di tali coppie saranno elencate le action contenute nel codice SLICC e relative alle transition corrispondenti.

Tabella 2

	Ifetch	Load	Store	Replacement
M	h,k	h,k	hh,k	i,d,zz / MI
E	h,k	h,k	hh,k / M	i,de,zz / MI
S	h,k	h,k	i,c,k / IM	ff,zz / I
I	i,gi,a,k / IE	i,gd,b,k / IE	i,gd,c,k / IM	ff,zz
MS	zz	zz	zz	zz / MI
MI	zz	zz	zz	zz
IM	zz	zz	zz	zz / IMI
IE	zz	zz	zz	zz / IEI
IMS	zz	zz	zz	zz / IMI
IMI	zz	zz	zz	zz
IES	zz	zz	zz	zz / IEI
IEI	zz	zz	zz	zz

Transizioni scatenate nella L1Cache dalle richieste del processore e dal rimpiazzamento

Tabella 3

	GET_INSTR	GETS	GETX	DataShared	DataExclusive	WBAck
M	i,ds,l / MS	i,ds,l / MS	i,d,l / MI	-	-	-
E	i,da,l / MS	i,da,l / MS	i,de,l / MI	-	-	-
S	l	l	ff,l / I	-	-	-
I	l	l	ff,l	-	-	-
MS	-	-	l / MI	-	-	s,l / S
MI	l	l	l	-	-	s,ff,l / I
IM	l / IMS	l / IMS	l / IMI	-	u,hh,s,l / M	-
IE	l / IES	l / IES	l / IEI	u,h,s,l / S	u,h,s,l / E	-
IMS	-	-	l / IMI	-	u,hh,ds,l	s,l / S
IMI	l	l	l	-	u,hh.d,l	s,ff,l / I
IES	-	-	l / IEI	u,h,da,l	u,h,da,l	s,l / S
IEI	l	l	l	u,h,de,l	u,h,de,l	s,ff.l / I

Transizioni scatenate nella L1Cache dai messaggi provenienti dalla L2Cache Privata

Tabella 4

GET_INSTR	GETS	GETX	ACCEPTS	PUTS	EJECT	PUTX
he,r,k	he,r,k	he,r,k	-	-	n,w,k	n,v,w,k
he,r,k	he,r,k	he,r,k / M	-	-	n,w,k	n,v,w,k / M
hs,k	hs,k	i,c,k / IM	w,k	-	w,k	-
i,g,n,a,k / IE	i,g,n,b,k / IE	i,g,n,c,k / IM	w,k	-	w,k	-
hs,k	hs,k	zz	w,k	-	w,k	-
zz	zz	zz	w,k	-	w,k	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	n,w,ee,ds,k / MS	n,v,w,ee,ds,k / MS	n,w,ee,ds,k / MS	n,v,w,ee,ds,k / MS
-	-	-	n,w,ee,gg,d,k / MI	n,v,w,ee,gg,d,k / MI	n,w,ee,gg,d,k / MI	n,v,w,ee,gg,d,k / MI
hs,k	hs,k	zz	w,k	v,w,ds,k	w,k	v,w,ds,k
zz	zz	zz	w,k	v,w,d,k	w,k	v,w,d,k
hs,k	hs,k	zz	w,k	-	w,k	-

Transizioni scatenzate nella L2Cache dai messaggi provenienti dalla L1Cache

Tabella 5

	Fwd_GET_INSTR	Fwd_GET_INSTR_I	Fwd_GETS	Fwd_GETS_I	Fwd_GETX	Fwd_GETX_I	Invalidate
M	i,ys,j,l / MSS	i,es,ds,l / MS	i,ys,j,l / MSS	i,es,ds,l / MS	i,yx,j,l / MSI	i,ex,d,l / MI	-
E	i,ys,j,l / MSS	i,es,da,l / MS	i,ys,j,l / MSS	i,es,da,l / MS	i,yx,j,l / MSI	i,ex,de,l / MI	-
S	-	-	-	-	-	-	m,t,ff,l / I
I	-	-	-	-	-	-	t,ff,l
MS	-	es,l	-	es,l	-	ex,j,l / MI	-
MI	-	es,l	-	es,l	-	ex,l	-
IM	-	ys,j,l / IMS	-	ys,j,l / IMS	-	yx,j,l / IMI	t,l
IE	-	ys,j,l / IES	-	ys,j,l / IES	-	yx,j,l / IEI	m,t,l / IEI
MSS	ys,l	-	ys,l	-	yx,j,l / MSI	-	-
MSI	ys,l	-	ys,l	-	yx,l	-	-
IMS	-	ys,l	-	ys,l	-	yx,j,l / IMI	t,l
IMI	-	ys,l	-	ys,l	-	yx,l	t,l
IES	-	ys,l	-	ys,l	-	yx,j,l / IEI	-
IEI	-	ys,l	-	ys,l	-	yx,l	t,l

Transizioni scanetate nella L2Cache dai messaggi provenienti dalla DirectoryCache (1)

Tabella 6

	DataShared	DataExclusive	Ack	LastAck	WBAck	Replacement	Replacement_I
M	-	-	-	-	-	i,m,zz / MSI	i,d,zz / MI
E	-	-	-	-	-	i,m,zz / MSI	i,de,zz /MI
S	-	-	-	-	-	-	m,ff,zz / I
I	-	-	-	-	-	ff,zz	ff,zz
MS	-	-	-	-	s,l / S	-	m,zz / MI
MI	-	-	-	-	s,ff,l / I	-	zz
IM	u,p,o	u,he,r,s,o / M	q,o	he,r,s,o / M	-	-	m,zz / IMI
IE	u,hs,s,o / S	u,he,r,s,o / E	-	-	-	-	m,zz / IEI
MSS	-	-	-	-	-	m,zz / MSI	-
MSI	-	-	-	-	-	zz	-
IMS	u,p,o	u,he,o	q,o	he,o	ee,s,l / S	-	m,zz / IMI
IMI	u,p,o	u,he,o	q,o	he,o	ee,gg,s,ff,l / I	-	zz
IES	-	u,hs,da,o	-	-	ee,s,l / S	-	m,zz / IEI
IEI	u,he,de,o	u,he,de,o	-	-	ee,gg,s,ff,l / I	-	zz

Transizioni scanetate nella L2Cache dai messaggi provenienti dalla DirectoryCache (2) e dai rimpiazzamenti

Tabella 7

	GET_INSTR	GETS	GETX	PUTX_Owner	PUTS_Owner	EJECT_Owner	ACCEPTS_Owner	WB_NotOwner
P	a,d,j / PS	a,d,j / PS	d,f,j	u,q,n,p,j / I	-	u,n,p,j / I	-	n,j
S	a,bs,j	a,bs,j	u,bs,f,h,g,j / P	-	-	-	-	n,j
I	f,bx,j / P	f,bx,j / P	f,bx,j / P	-	-	-	-	n,j
PS	a,d,j	a,d,j	u,d,f,h,g,j / P	u,q,n,p,j / S	a,q,n,p,j / S	u,n,p,j / S	a,n,p,j / S	n,j
R	v,c,zz / F	v,c,zz / F	v,c,zz / F	v,c,zz / F	v,c,zz / F	v,c,zz / F	v,c,zz / F	-
F	zz	zz	zz	zz	zz	zz	zz	zz

Transizioni scanenate nella DirectoryCache da messaggi provenienti dalla L2Cache

Tabella 8

	Replacement	DirBlock_P	DirBlock_S	DirBlock_I	DirBlock_PS
P	r,ff,zz / R	-	-	-	-
S	r,ff,zz / R	-	-	-	-
I	r,ff,zz / R	-	-	-	-
PS	r,ff,zz / R	-	-	-	-
R	-	-	-	-	-
F	zz	l,jj / P	l,jj / S	l,jj / I	l,jj / PS

Transizioni scatenate nella DirectoryCache dai messaggi provenienti dalla Directory

Tabella 9

	GET	PUT_P	PUT_S	PUT_I	PUT_PS	GETX	GETS	PUTX
C	-	l,j / P	l,j / S	l,j / I	l,j / PS	ce,j	cs,j	m,j
P	sp,j / C	-	-	-	-	-	-	-
S	ss,j / C	-	-	-	-	-	-	-
I	si,j / C	-	-	-	-	-	-	-
PS	sps,j / C	-	-	-	-	-	-	-

Transizioni scatenate nella Directory dai messaggi provenienti dalla DirectoryCache

Tabella 10

ID	Descrizione
a	Invia un messaggio di tipo GET_INSTR verso la Cache L2.
b	Invia un messaggio di tipo GETS verso la Cache L2.
c	Invia un messaggio di tipo GETX verso la Cache L2.
d	Invia un messaggio di tipo PUTX verso la Cache L2.
da	Invia un messaggio di tipo ACCEPTS verso la Cache L2
de	Invia un messaggio di tipo EJECT verso la Cache L2.
ds	Invia un messaggio di tipo PUTS verso la Cache L2.
ff	Dealloca un blocco dalla ICache o dalla Dcache.
gd	Alloca spazio nella L1-DCache se il blocco non è presente in cache.
gi	Alloca spazio nell L1-ICache se il blocco non è presente in cache.
h	Comunica al sequencer che la load è stata completata.
hh	Comunica al sequencer che la store è stata completata.
i	Alloca una TBE per il blocco corrente, per il quale è iniziata una transizione di stato.
k	Elimina la richiesta in testa alla MandatoryQueue.
l	Estrae il messaggio in testa alla Coda dei Messaggi provenienti dalla Cache L2.
s	Dealloca la TBE per il blocco corrente.
u	Scrive la copia del blocco di memoria nel blocco di cache precedentemente allocato.
zz	Riaccoda il messaggio nella mandatoryQueue.

Azioni della L1Cache

Tabella 11

ID	Descrizione
a	Inserisco l'ID del richiedente nella lista degli sharers.
bx	Invia un messaggio di tipo GETX verso la Directory.
bs	Invia un messaggio di tipo GETS verso la Directory.
c	Invia un messaggio di tipo GET verso la Directory.
d	Instrada la richiesta attuale sulla fwdNetwork verso l'Owner.
f	Imposta il mittente del messaggio in ingresso come owner del blocco.
ff	Dealloca il blocco di Directory Cache.
g	Svuota la lista degli Sharers per il blocco.
h	Invia un messaggio di tipo INV verso tutti gli sharers del blocco.
j	Estrae il messaggio in testa alla RequestQueue.
jj	Estrae il messaggio in testa alla ResponseQueue.
l	Scrive nella DirectoryCache il blocco recuperato dalla Directory (Fetch).
n	Invia un messaggio WB_Ack, in risposta ad una ACCEPTS/PUTS/PUTX/EJECT.
p	Toglie la ownership all'owner corrente, riassegnandola alla Directory.
q	Invia un messaggio di tipo PUTX verso la Directory.
r	Invia alla Directory il blocco presente nella DirectoryCache (Rimpiazzamento).
u	Toglie il mittente dell'in_msg dalla lista degli Shareres per il Blocco.
v	Alloca spazio per un blocco nella Directory Cache.
zz	Riaccoda il messaggio nella Request Queue.

Azioni della DirectoryCache

Tabella 12

ID	Descrizione
a	Invia un messaggio di tipo GET_INSTR verso la DirectoryCache.
b	Invia un messaggio di tipo GETS verso la DirectoryCache.
c	Invia un messaggio di tipo GETX verso la DirectoryCache.
d	Invia un messaggio di tipo PUTX verso la DirectoryCache.
da	Invia un messaggio di tipo ACCEPTS verso la DirectoryCache.
de	Invia un messaggio di tipo EJECT verso la DirectoryCache.
ds	Invia un messaggio di tipo PUTS verso la DirectoryCache.
ee	Legge dalla relativa TBE gli ID dei nodi cui bisogna inoltrare la copia del blocco ed accoda il messaggio.
es	Invia la copia del blocco verso una cache remota richiedente con un messaggio DATA_SHARED.
ex	Invia la copia del blocco verso una cache remota richiedente con un messaggio DATA_EXCLUSIVE.
ff	Dealloca un blocco di Cache, impostandolo a Invalid
g	Alloca spazio in L2-Cache se il blocco non è presente in cache.
gg	Legge dalla relativa TBE l'ID della cache remota cui bisogna inoltrare la copia del blocco ed accoda il messaggio.
he	Invia la copia del blocco verso la L1 Cache con un messaggio DATA_SHARED.
hs	Invia la copia del blocco verso la L1 Cache con un messaggio DATA_EXCLUSIVE.
i	Alloca una TBE per il blocco corrente, per il quale è iniziata una transizione di stato.
j	Inoltra le GET_INSTR/GETS/GETX/INV provenienti dalla Directory verso la Cache L1.
k	Estrae il messaggio in testa alla coda del messaggi provenienti dalla Cache L1.
l	Estrae il messaggio in testa alla coda delle richieste di forwarding.
m	Invia un messaggiodi tipo GETX verso la Cache L1.
n	Imposta il flag L1Invalid a true.
o	Estrae il messaggio in testa alla coda delle risposte.
p	Somma il numero di pending ack del messaggio a quello nel relativo campo della TBE.
q	Decrementa di uno il numero di pending ack nel relativo campo della TBE.
r	Imposta il flag L1Invalid a false.
s	Dealloca la TBE per il blocco corrente.
t	Invia un messaggio ACK di risposta ad una richiesta di invalidazione.
u	Srive la copia del blocco di memoria nel blocco di cache precedentemente allocato.
v	Scrive nella Cache L2 il blocco rimpiazzato dalla Cache L1.
w	Invia un messaggio WB_ACK alla Cache L1.
ys	Aggiunge nel campo ForwardGETS_IDs della TBE l'ID del nodo che ha richiesto il blocco non esclusivo
yx	Scrive nel campo ForwardGETX_ID del TBE l'ID del nodo che ha richiesto il blocco esclusivo assieme al numero di pending ack
zz	Riaccoda il messaggio nella L2-from-L1 Queue.

Azioni della L2Cache

Tabella 13

ID	Descrizione
ce	Invia un messaggio di tipo DATA_EXCLUSIVE al richiedente.
cs	Invia un messaggio di tipo DATA_SHARED al richiedente.
j	Estrae il messaggio in testa all RequestQueue.
l	Scrive il blocco directory in memoria principale.
m	Scrive il blocco di memoria in memoria principale.
si	Invia alla DirectoryCache il blocco di directory nello stato I.
sp	Invia alla DirectoryCache il blocco di directory nello stato P.
sps	Invia alla DirectoryCache il blocco di directory nello stato PS.
ss	Invia alla DirectoryCache il blocco di directory nello stato S.

Azione della Directory

Riferimenti

[1] K. Olukotun, B.A. Nayefeh, L. Hammond, K.Wilson and K.Chang, “The case for a single-chip Multiprocessor” Proc. Of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, pp. 241-251, Oct 1996.

[2] L. Hammond, B.A. Nayefeh and K. Olukotun, “A single-chip Multiprocessor” IEEE Computer, 30(9), pp. 79-85, Sept. 1997.

[3] K. Krewell, “UltraSparc IV Mirrors Predecessors”, Microprocessor Report, pp. 1-3, Nov. 1997.

[4] C. McNairy and R. Bhatia, “Montecito: A Dual-Core Dual-Thread Itanium Processor” IEEE Micro, 25(2), pp. 10-20, Mar./Apr. 1997.

[5] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer and J. Joyner, “Power5 System Architecture” IBM Journal of Research and Development, 49(4), pp. 505-522, 2005.

[6] P. Kongetira, K. Aingaran and K. Olukotun, “Niagara: a 32-way multithreaded Sparc processor” IEEE Micro, 25(2), pp. 21-29, Mar. 2005.

[7] B.M. Beckmann and D.A. Wood, “Managing Wire Delay in Large Chip-Multiprocessor Caches” Proc. Of the 37th annual IEEE/ACM Int. Symp. on Microarchitecture, Portland, OR, USA, pp. 319-330, Dec. 2004.

[8] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer and A. Kumar, “CMP implementation in systems based on the Intel Core Duo processor” Intel Technology Journal, 10(2), pp. 99-107, 2006.

[9] Z. Chisti, M.D. Powell and T.N. Vijaykumar, “Optimizing Replication, Communication and Capacity Allocation in CMPs” Proc. of the 32nd annual Int. Symp. on Computer Architecture, Madison, WI, USA, pp. 357-368, June 2005.

[10] J. Chang and G.S. Sohi, “Cooperative Caching for Chip Multiprocessors” Proc. of the 33rd annual Int. Symp. on Computer Architecture, Boston, MA, USA, pp. 264-276, June 2006.

[11] M. Zhang and K. Asanovic, “Victim Delay in Tiled Chip Multiprocessors” Proc. of the 32nd annual Int. Symp. on Computer Architecture, Madison, WI, USA, pp. 336-345, June 2005.

[12] R. Ho, K.W. Mai and M.A. Horowitz, “The future of wires” Proc. of the IEEE, 89(4), pp. 490-504, April 2001.

[13] C. Kim, D. Burger and S.W. Keckler, “An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches” Proc. of the 10th Int. Conf. On Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, pp. 211-222, Oct. 2002.

[14] J. Kuh, C. Kim, H. Shafi, L. Zhang, D. Burger, S.W. Keckler, “A NUCA substrate for flexible CMP cache sharing” Proc. of the 19th annual Int. Conf. on Supercomputing, Cambridge, MA, USA, pp. 31-40, June 2005.

[15] J. Duato, S. Yalamanchili and L. Ni, Interconnection Network an Engineering Approach. San Francisco, CA, Morgan Kauffmann, Elsevier, 2003.

[16] W.J. Dally and B. Towels, Principles and Practices of Interconnection Networks. San Francisco, CA, Morgan Kauffmann, Elsevier, 2004.

[17] B.M. Beckmann, M.R. Marty, D.A. Wood, “ASR: Adaptive Selective Replication for CMP Caches” Proc. of the 39th annual IEEE/ACM Int. Symp. on Microarchitecture , Orlando, FL, USA, pp. 443-454, Dec. 2006.

[18] C. Bienia, S. Kumar, J. Pal Singh and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications” Proc. of the 17th Int. Conf. on Parallel Architecture and Compilation Techniques, Toronto, Canada, pp. 72-81, Oct. 2008.

[19] K. Gharachorloo, M. Sharma, S. Steely and S. Van Doren, “Architecture and Design of AlphaServer GS320” Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, pp. 13-24, Nov. 2000.

[20] P. Foglia, F. Panicucci, C.A. Prete and M. Solinas, “Investigating design tradeoffs in S-NUCA based CMP systems” Proc. of the 5th annual workshop on Unique Chips and Systems, Boston, MA, USA, pp. 53-60, 2009.

[21] P. Foglia, F. Panicucci, C.A. Prete and M. Solinas, “An evaluation of behaviors of S-NUCA CMPs running scientific workload” Proc. of the 12th Euromicro conf. on Digital Systems Design, Patras, Greece, Aug. 2009.

[22] Q. Lu, U. Bondhugula, S. Krishnamoorthy, P. Sadayappan, J. Ramanujam, Y. Chen, H. Lin and T. Ngai, “A compile-time data locality optimization frame work for NUCA chip multiprocessore” Technical Report, OSU-CISRC-6/08-TR29.

[23] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli and C.A. Prete, “A power-efficient migration mechanism for D-NUCA caches” Proc. of the Design, Automation and Test in Europe 09, Nice, France, pp. 598-601, Apr. 2009.

[24] Virtutech Simics, http://www.virtutech.com

[25] Wisconsin Multifacet GEMS Simulator, http://www.cs.wisc.edu/gems/

[26] S. Thoziyoor, N. Muralimanohar, J.H. Ahn and N.P. Jouppi, “CACTI 5.1” Technical Report, HP Laboratories, Palo Alto, CA, USA, April 2008.

[27] Predictive Technology Model (PTM), http://www.eas.asu.edu/~ptm/

[28] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, “The SPLASH-2 programs: characterization and methodological considerations” Proc. of the 22nd Int. Symp. on Computer Architecture, S. Margherita Ligure, Italy, pp. 24-36, June 2005.

[29] P. Foglia, F. Panicucci, C.A. Prete and M. Solinas, “Analysis of performance dependencies in NUCA-based CMP systems” Proc. of the 21st Int. Symp. on Computer Architecture and High Performance Computing, Sao Paulo, Brazil, pp. 49-56, Oct. 2009.

[30] J.L. Henning, “SPEC CPU 2000: Measuring CPU Performance in the New Millennium”, IEEE Computer, 33(7), pp. 28-35, July 2000.

[31] P. Foglia, C.A. Prete, M. Solinas, G. Monni, “Re-NUCA: Boosting CMP performance through block replication”, 14th Int.l Conf. On Digital Sysyem Design (DSD2010), Lille, France, 1-3 Sept. 2010.